Dynamic C 5.x

Integrated C Development System

Function Reference

Revision 3

Z-World • Dynamic C 5.x

Function Reference • Part Number 019-0002-03
Revision 3 • 021-0005-03 • Printed in U.S.A.
Last revised by TI • August 21, 1998

Copyright

Trademarks

Notice to Users

When a system failure may cause serious consequences, protecting life and property against such consequences with a backup system or safety device is essential. The buyer agrees that protection against consequences resulting from system failure is the buyer's responsibility.

This device is not approved for life-support or medical systems.

Company Address

Z-World
2900 Spafford Street
Davis, California 95616-6800 USA

Telephone:	(530) 757-3737
Facsimile:	(530) 753-5141
24-Hour FaxBack:	(530) 753-0618
Web Site:	http://www.zworld.com
E-Mail:	zworld@zworld.com

TABLE OF CONTENTS

ABOUT THIS MANUAL

Z-World customers develop software for their programmable controllers using Z-World's Dynamic C development system running on an IBM-compatible PC. The controller is connected to a COM port on the PC, usually COM2, which by default operates at 19,200 bps.

The Standard version of Dynamic C is suitable for programs up to 80K, with limited access to extended memory. The Deluxe version supports programs with up to 512K in ROM (code and constants) and 512K in RAM (variable data), with full access to extended memory.

The Three Manuals

Dynamic C is documented with three reference manuals:

- Dynamic C Technical Reference
- Dynamic C Application Frameworks
- Dynamic C Function Reference.

The Technical Reference manual describes how to use the Dynamic C development system to write software for a Z-World programmable controller.

The Application Frameworks manual discusses various topics in depth. These topics include the use of the Z-World real-time kernel, costatements, function chaining, and serial communication.

This manual contains descriptions of all the function libraries on the Dynamic C disk and all the functions in those libraries.

Please read release notes and updates for late-breaking information about Z-World products and Dynamic C.

Assumptions

Assumptions are made regarding the user's knowledge and experience in the following areas:

- Understanding of the basics of operating a software program and editing files under Windows on a PC.
- Knowledge of the basics of C programming. Dynamic C is not the same as standard C.

  For a full treatment of C, refer to the following texts:

  The C Programming Language by Kernighan and Ritchie (published by Prentice-Hall).

  and/or

  C: A Reference Manual by Harbison and Steel (published by Prentice-Hall).

- Knowledge of basic Z80 assembly language and architecture.

  For documentation from Zilog, refer to any of the following texts:

  Z180 MPU User's Manual
  Z180 Serial Communication Controllers
  Z80 Microprocessor Family User's Manual

Acronyms

Table 1 lists the acronyms that may be used in this manual.

Table 1. Acronyms

Acronym	Meaning
EPROM	Erasable Programmable Read-Only Memory
EEPROM	Electronically Erasable Programmable Read-Only Memory
NMI	Nonmaskable Interrupt
PIO	Parallel Input/Output Circuit (Individually Programmable Input/Output)
PRT	Programmable Reload Timer
RAM	Random Access Memory
RTC	Real-Time Clock
SIB	Serial Interface Board
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter

Conventions

Table 2 lists and defines typographic conventions that may be used in this manual.

Table 2. Typographic Conventions

Example	Description
`While`	Courier font (bold) indicates a program, a fragment of a program, or a Dynamic C keyword or phrase.
`// IN-01…`	Program comments are written in Courier font, plain face.
Italics	Indicates that something should be typed instead of the italicized words (e.g., in place of *filename*, type a file's name).
Edit	Sans serif font (bold) signifies a menu or menu selection.
. . .	An ellipsis indicates that (1) irrelevant program text is omitted for brevity or that (2) preceding program text may be repeated indefinitely.
[ ]	Brackets in a C function's definition or program segment indicate that the enclosed directive is optional.
< >	Angle brackets occasionally enclose classes of terms.
a | b | c	A vertical bar indicates that a choice should be made from among the items listed.

Programming Abbreviations

This manual uses these programming abbreviations for convenience.

- **`uint`** means **`unsigned integer`**
- **`ulong`** means **`unsigned long`**

These abbreviations are not standard C keywords, and will not work in an application unless they are first declared with **`typedef`** or **`#define`** as in the examples shown below.

```
typedef unsigned int uint
```

or

```
#define ulong unsigned long
```

Icons

Table 3 displays and defines icons that may be used in this manual.

Table 3. Icons

Icon	Meaning
👓	Refer to or see
☎	Please contact
⚠	Caution
✎	Note
⚡	High Voltage
TIP	Tip
FD	Factory Default

☎	For ordering information, call your Z-World Sales Representative at (530) 757-3737.

1

General Support Libraries

The libraries described in Chapter 1 include standard C string and math functions in addition to general support functions specific to Z-World's controllers.

Global Initialization

Global initialization is an important, but unclassifiable topic, and is described here. Your program can initialize variables and take initialization action (of any complexity) if you do the following:

1. Incorporate **`_GLOBAL_INIT`** segments in your functions:

```
void init_ios();

int my_func( void* thing ){
  int table[10],j;
  float x,y;
    ...
  segchain _GLOBAL_INIT{
    for( j=0; j<10; j++ ){ table[j] = 10-j; }
    x = y = 0.781;
    init-ios();
  }
    ...
}
```

2. Make a call to the function chain **`_GLOBAL_INIT`** at the start of main.

When your program starts (from scratch or because of a hardware reset) the call to **`_GLOBAL_INIT`** performs the initialization for all **`_GLOBAL_INIT`**s throughout your program (including libraries). The name **`_GLOBAL_INIT`** is not the name of a library function. However, there is a function **`GLOBAL-INIT`** in **`VDRIVER.LIB`**. If you call **`VdInit`**, i.e., you invoke the virtual driver, **`VdInit`** does global initialization for you. You need not do it yourself. The function **`uplc_init`** also calls **`_GLOBAL_INIT`**.

BIOS Functions

These functions reside in BIOS. The source code is provided for your convenience. To override BIOS function, use

```
#kill functionname
```

at the beginning of your user program and redefine the function.

- **`uint inport( uint port )`**

  Reads a value from the specified I/O port. This may be an internal Z180 register, or it may access external hardware. Refer to the controller reference manual for a list of I/O ports.

  The function returns the value from the I/O port in lower byte, and zero in upper byte.

- **`void outport( uint port, uint value )`**

  Writes **`value`** to I/O port. This may be an internal Z180 register, or it may access external hardware. Refer to your controller reference manual for a list of I/O ports.

- **`int ee_rd( int address )`**

  Reads value from EEPROM at specified address. The function returns EEPROM data (0–255) if successful. It returns a negative value if unable to read the EEPROM.

- **`int ee_wr( int address, char value )`**

  Writes value to EEPROM at specified address. The function returns 0 if successful. It returns a negative value if unable to write the EEPROM.

- **`void di()`**

  Disables interrupts. Use **`DI`** for better efficiency.

- **`void DI()`**

  Disables interrupts. Dynamic C expands this call inline.

- **`void ei()`**

  Enables interrupts. Use **`EI`** for better efficiency.

- **`void EI()`**

  Enables interrupts. Dynamic C expands this call inline.

- **`int iff()`**

  Returns the state of the Z180 interrupt mask. If zero, interrupts are off. Otherwise, interrupts are on.

- **`uint bit( void* address, uint bit )`**

  Reads the value of the specified **`bit`** at memory address. The **`bit`** may be from 0 to 31. Use **`BIT`** (upper case) for inline expansion of this call. This is equivalent to the following expression:

  ```
  (*(long*)address >> bit) & 1
  ```

  The function returns 1 if specified **`bit`** is set; 0 if **`bit`** is clear.

- **`uint BIT( void *address, uint bit )`**

  Reads the value of the specified **`bit`** at memory address. The bit may be from 0 to 31. Dynamic C will attempt to expand this call inline. This is equivalent to the following expression:

  ```
  (*(long*)address >> bit) & 1
  ```

  The function returns 1 if specified **`bit`** is set, and 0 if **`bit`** is clear.

- **`void set( void *address, uint bit )`**

  Sets the specified **`bit`** at memory address to 1. The **`bit`** may be from 0 to 31. Use **`SET`** (upper case) for inline expansion of this call. This is equivalent to the following expression:

  ```
  *(long*)address |= 1L << bit
  ```

- **`void SET( void *address, uint bit )`**

  Sets the specified **`bit`** at memory address to 1. The **`bit`** may be from 0 to 31. Dynamic C will attempt to expand this call inline. This is equivalent to the following expression:

  ```
  *(long*)address |= 1L << bit
  ```

- **`void res( void *address, uint bit )`**

  Clears specified **`bit`** at memory address to 0. **`bit`** may be from 0 to 31. Use **`RES`** (upper case) for inline expansion of this call. This is equivalent to the following expression:

  ```
  *(long*)address &= ~(1L << bit)
  ```

- **`void RES( void *address, uint bit )`**

  Clears specified **`bit`** at memory address to 0. **`bit`** may be from 0 to 31. Dynamic C will attempt to expand this call inline. This is equivalent to the following expression:

  ```
  *(long*)address &= ~(1L << bit)
  ```

- **`uint IBIT( uint port, uint bit )`**

  Reads the I/O port and returns the value of the specified **`bit`**. The bit may be from 0 to 7. The port may be an internal Z180 register, or it may access external hardware. Refer to your controller reference manual for a list of I/O ports. The function returns 1 if the specified **`bit`** is set, and 0 if the **`bit`** is clear.

- **`void ISET( uint port, uint bit )`**

  Sets the specified **`bit`** of the I/O port to 1. The **`bit`** may be from 0 to 7. The port may be an internal Z180 register, or it may access external hardware. The function generates code like the following:

  ```
      in   a,(c)
      set  bit,a
      out  (c),a
  ```

  Refer to the controller reference manual for a list of I/O ports.

- **`void IRES( uint port, uint bit )`**

  Resets the specified bit of the I/O port to 0. The **`bit`** may be from 0 to 7. The **`port`** may be an internal Z180 register, or it may access external hardware. The function generates code like the following:

  ```
  in   a,(c)
  set  bit,a
  out  (c),a
  ```

  Refer to the controller reference manual for a list of I/O ports.

- **`void hitwd()`**

  "Hits" the watchdog timer, postponing a hardware reset for approximately 1.2–1.6 seconds (the value depends on hardware). Unless the watchdog timer is disabled, the program must call this function periodically. Otherwise, the controller resets automatically. This allows the controller to recover from errors that cause the program to enter an infinite loop. If the virtual driver is enabled, it will call **`hitwd`** in the background but provide virtual watchdogs in its place. See **`VdWdogHit`** for more information. For information about setting jumpers to enable/disable the watchdog (not available on all boards), refer to the controller reference manual.

- **`int wderror()`**

  Determines if the previous reset was caused by the watchdog timer. This feature is not available on all boards. Refer to the controller reference manual for more information.

  The function returns a positive non-zero value if the watchdog caused the last reset and zero if not. It returns a negative value if the feature is not supported.

- **`void intrmode_0()`**

  Sets Z180 interrupt mode to 0. The default mode for Dynamic C is Mode 2. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

- **`void intrmode_1()`**

  Sets Z180 interrupt mode to 1. The default mode for Dynamic C is Mode 2. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

  The function returns None.

- **`void intrmode_2()`**

  Sets Z180 interrupt mode to 2. This is the default mode for Dynamic C. Do not select another mode unless the interrupts for all peripheral devices using Mode 2 interrupts have been disabled.

- **`void runwatch()`**

  Allows Dynamic C to update watch expressions. Calling **`runwatch`** periodically enables evaluation of watch expressions while the program is running. Watch expressions are always evaluated when the program is stopped.

- **`int kbhit()`**

  Detects keystrokes in the Dynamic C **STDIO** window. The function returns non-zero if a key has been pressed, and zero otherwise.

- **`void exit( int exitcode )`**

  Stops the program and returns **`exitcode`** to Dynamic C. Dynamic C uses code values above 128 for run-time errors. When not debugging, this function causes a watchdog time-out if the watchdog is enabled.

  The function does not return.

- **`uint sysclock()`**

  Returns the system clock speed in units of 1200 Hz. Some common clock speeds and the corresponding **`sysclock`** values are listed below.

6.144 MHz	0x1400 (5120)	9.126 MHz	0x1E00 (7680)
12.288 MHz	0x2800 (10,240)	18.432 MHz	0x3C00 (15,360)

  The function returns clock speed / 1200.

- **`int powerlo()`**

  It is possible for the supply voltage to drop low enough to generate a power-fail interrupt, but then return to normal without ever dropping low enough to reset the board. Call this routine from an NMI (power-fail) interrupt handler to determine if power has returned. Refer to the controller reference manual to find out whether this feature is supported. The function returns 1 if voltage is below the NMI level, and 0 otherwise.

MATH.LIB

The Z-World standard library contains floating-point functions in addition to I/O functions. Normal mathematical limitations apply to these functions, and any function generating a value outside the accepted floating-point range (about 10^{38} to -10^{38}) will result in an overflow error. Infinity is defined as INF in **`DC.HH`**.

Trigonometric functions such as tan(x) generally accept arguments in radians. Certain trig functions may fail if their argument is too large. Any angle may be normalized to fall within the range $[-\pi, \pi]$ without loss of accuracy.

- **`int abs( int x )`**

  Computes the absolute value of an integer argument.

- **`float acos( float x )`**

  Computes the arccosine of **`x`**. The value of **`x`** must be between –1 and +1. If **`x`** is out of bounds, the function returns 0 and signals a domain error.

- **`float acot( float x )`**

  Computes the arccotangent of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float acsc( float x )`**

  Computes the arccosecant of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float asec( float x )`**

  Computes the arccosecant of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float asin( float x )`**

  Computes the arcsine of **`x`**. The value of **`x`** must be between –1 and +1. If **`x`** is out of bounds, the function returns 0 and signals a domain error.

- **`float atan( float x )`**

  Computes the arctangent of **`x`**. The value of **`x`** must be between –INF and +INF.

- **`float atan2( float y, float x )`**

  Computes the arctangent of **`y/x`**. If both **`y`** and **`x`** are zero, the function returns 0 and signals a domain error. Otherwise the result is returned as follows:

angle	$x \neq 0, y \neq 0$
PI/2	$x = 0, y > 0$
–PI/2	$x = 0, y < 0$
0	$x > 0, y = 0$
PI	$x < 0, y = 0$

- **`float ceil( float x )`**

  Returns the smallest integer greater than or equal to **`x`**.

- **`float cos( float x )`**

  Computes the cosine of **`x`**.

- **`float cosh( float x )`**

  Computes the hyperbolic cosine of **`x`**. If |**`x`**| > 89.8 (approx.), the function returns INF and signals a range error.

- **`float deg( float x )`**

  Returns angle in degrees for angle **`x`** given in radians.

- **`float rad( float x )`**

  Returns angle in radians for angle **`x`** given in degrees.

- **`float exp( float x )`**

  Returns the value of e^x. If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < –89.8 (approx.), the function returns 0 and signals a range error.

- **`float fabs( float x )`**

  Computes the absolute value of **`x`**. The function returns **`x`** if **`x`** ≥ 0; otherwise it returns –**`x`**.

- **`float floor( float x )`**

  Computes the largest integer less than or equal to the given number.

- **`float fmod( float x, float y )`**

  Returns the *remainder* of **`x`** with respect to **`y`**, that is, the remaining part of **`x`** after all multiples of **`y`** have been removed. For example, if **`x`** is 22.7 and **`y`** is 10.3, the integral division result is 2. Then the remainder = 22.7 – 2 × 10.3 = 2.1.

- **`float frexp( float x, int *n )`**

  This function splits **`x`** into a fraction and exponent ($f \times 2^n$). The function returns the exponent in the integer **`*n`** and the fraction (between 0.5 and 0.999...) as the function result.

- **`long labs( long x )`**

  Computes the absolute value of long integer **`x`**. The function returns **`x`** if x ≥ 0; otherwise it returns –**`x`**.

- **`float ldexp( float x, int n )`**

  Computes **`x`***(radix****`n`**), where **`n`** is an integer and 0.5 ≤ **`x`** < 1.0.

- **`float log( float x )`**

  Computes the natural logarithm (base e) of **`x`**. The function returns –INF and signals a domain error when **`x`** ≤ 0.

- **`float log10( float x )`**

  Computes the base 10 logarithm of **`x`**. The function returns –INF and signals a domain error when **`x`** ≤ 0.

- **`float modf( float x, int *n )`**

  Splits **`x`** into an integer part and fractional part, f + **`n`**, where **`n`** is the integer and f satisfies $|f| < 1.0$. The function returns the integer part in **`*n`** and returns the fractional part as the function result.

- **`float poly( float x, int n, float c[] )`**

  Computes a polynomial value by Horner's method. The term **`x`** is the variable of the polynomial, **`n`** is the order of the polynomial, and **`c`** is an array containing the coefficients of each power of **`x`** . For example, for the fourth-order polynomial

  $10x^4 - 3x^2 + 4x + 6$

  `n` would be 4 and the coefficients would be

  ```
  c[4] = 10.0
  c[3] = 0.0
  c[2] = -3.0
  c[1] = 4.0
  c[0] = 6.0
  ```

- **`float pow( float x, float y )`**

  Returns $\mathbf{x}^{\mathbf{y}}$.

- **`float pow10( float x )`**

  Returns 10^x.

- **`float sin( float x )`**

  Computes the sine of **`x`**.

- **`float sinh( float x )`**

  If **`x`** > 89.8 (approx.), the function returns INF and signals a range error. If **`x`** < –89.8 (approx.), the function returns –INF and signals a range error.

- **`float sqrt( float x )`**

  Computes the square root of **`x`**.

- **`float tan( float x )`**

  Return the tangent of **`x`**, where –8×PI ≤ **`x`** ≤ +8×PI. If **`x`** is out of bounds, the function returns 0 and signals a domain error. If the value of **`x`** is too close to a multiple of 90° (PI / 2) the function returns INF and signals a range error.

- **`float tanh( float x )`**

  Returns the hyperbolic tangent of **`x`**. If **`x`** > 49.9 (approx.), the function returns INF and signals a range error. If **`x`** < –49.9 (approx.), the function returns –INF and signals a range error.

- **`float _pow10( int exp )`**

  Computes integral powers of 10 (10^{exp}).

STDIO.LIB

The following functions address the standard I/O window in Dynamic C, which is used for debugging.

- **`char *gets( char* s )`**

  This function waits for a string terminated by a ‹CR› (carriage return) to be typed. It does not return until a ‹CR› is typed in the **STDIO** window. However, the string returned is null terminated. The function returns the typed string at the location identified by the pointer **`s`**. Make sure the storage is big enough for the string and that only one process calls this function at a time.

- **`char getchar( void )`**

  This function waits (in an idle loop) for a character to be typed from the **STDIO** window in Dynamic C. Make sure only one process calls this function at a time.

- **`int puts( char *s )`**

  This function writes the string, identified by pointer **`s`**, in the **STDIO** window in Dynamic C. The **STDIO** window will interpret any escape code sequences contained in the string. Make sure only one process calls this function at a time. The function returns 1 if successful.

- **`void putchar( int ch )`**

  Writes a single character (the lower 8 bits of **`ch`**) to **STDIO**. Make sure only one process calls this function at a time.

- **`void sprintf( char *buffer, char *format, ... )`**

  An analog of standard function **`printf`**, this function takes a "format" string (**`*format`**), and a variable number of value arguments to be formatted. It formats the arguments and places the formatted string in **`*buffer`**. Make sure that:

  1. There are enough arguments after **`format`** to fill in the format parameters in the format string.
  2. The types of arguments after **`format`** match the format fields in **`format`**.
  3. **`buffer`** is large enough to hold the longest possible formatted string.

For example,

```
sprintf( buffer,"%s=%x","Variable x",256 )
```

should put the string “Variable x=100” into **buffer**. This function can be called by processes of different priorities.

The functions **printf()** and **sprintf()** are not reentrant. The function **doprnt** implements **printf** and **sprintf**, and uses the character output function **putc** specified by the programmer. These functions accept format strings and a variable number of parameters whose values are to be printed according to the format, for example,

```
printf( "Summary for %s:\n", person );
printf( "  Age: %d, Income: $%8.2f", age, income );
```

The first statement prints a character string. The **%s** in the format tells the function where and how to print the character string.

The second statement prints two numbers, an integer **age** and a float **income**. The **%d** in the format tells the function where and how to print the integer: as a decimal string, free-formatted. The **%8.2f** in the format tells the function to print income as a floating value, with a field width of eight characters and two decimal places.

```
Summary for Sally Forth:
Age: 39, Income: $39587.02
```

The complete syntax of a field code is:

%[+|-][*width* [.*precision*]]*letter*

where

+ makes the value right-justified in its field, if a field width is specified.

- makes the value left justified in its field, if a field width is specified.

width is the field width. If not specified, the field width varies according to the value.

precision for floating-point values, that is, the number of digits to the right of the decimal.

letter selects the interpretation of the data according to the following list.

d decimal conversion (expects type **int**)

o octal conversion

x hex conversion

u unsigned decimal conversion (expects type **uint**)

c single **char** representation (expects type **char**)

s string (with null termination)

`e`	mantissa/exponent form of floating point (expects type **`float`**)
`f`	normal floating point (expects type **`float`**)
`g`	use e or f conversion, whichever is shorter. (expects **`float`**)
`l`	decimal conversion (expects type **`long`**)

- **`void printf( char *fmt, ... )`**

  This standard function accepts a variable number of value arguments, composes a formatted string from the values, and writes the formatted string to the **STDIO** window. Refer to the description of **`sprintf`** for details. Only one process should use this function at any time.

- **`void doprnt( int(*put)(),char *fmt,void* arg1 )`**

  This is the support routine behind all **`..printf`** routines. Passes a function **`put`** that outputs one byte. It will be called whenever **`doprnt`** outputs a character. The term **`fmt`** is the format string that specifies the output. The term **`arg1`** points to the first parameter to be used by the formatted string. The interpretation of the parameters depends on the format fields in the format string. This routine causes many math functions to be compiled and downloaded. This routine can be called from processes of different priorities.

- **`char *gtoa( ulong num, char *ibuf )`**

  This function uses **`_gltoa`** to output an unsigned long integer, **`num`**, to the character array **`*ibuf`**. The function returns a pointer to **`ibuf`**.

- **`char *ltoa( long num, char *ibuf )`**

  This function uses **`_gltoa`** to output a signed long integer, **`num`**, to the character array **`*ibuf`**. The function returns a pointer to **`ibuf`**.

- **`int gtoan( ulong num )`**

  This function returns the number of characters required to display a unsigned long integer, **`num`**.

- **`int ltoan( long num )`**

  This function returns the number of characters required to display a signed long integer, **`num`**.

- **`void pint( char flag, char code, int width, int(*put)(), int value )`**

  Writes a short integer value as a decimal string according to the user-specified single-character output procedure **`put`**. The term **`width`** specifies field width. If zero is specified, the field will be as wide as needed to represent **`value`**. The **`flag`**, if **`'-'`**, indicates that the field is left-justified. Otherwise, it is right-justified. If **`code`** is **`'d'`**, the function treats **`value`** as a signed integer, otherwise as an unsigned integer. The function prints all asterisks if the value does not fit in the field specified.

- **`void plint( char left, char code, int n1, int(*put)(), long num )`**

  This function has the same effect as **`pint`**, but accepts and prints a long integer.

- **`int ftoa( float f, char *buf )`**

  Converts the floating pointer number **`f`** to a character string **`*buf`**. The string will be no longer than 12 characters long. The character string only displays the mantissa up to 12 digits, with no decimal points. The function returns the exponent (base 10) that should be used to compensate for the missing decimal point. For example,

  ```
  ftoa(1.0,buf)
  ```

  generates the string “1000000000”, and returns –10. If **`f`** is 45.678, **`ftoa`** will generate the character string “45678” and return the integer exponent –3, indicating 45678×10^{-3}.

- **`void plhex( char left, int n1, int(*put)(), long num )`**

  Writes a long (signed or unsigned) integer in hex format. The term **`left`** specifies the padding character that goes to the left side of the actual number. If **`left`** is **`'-'`**, white space is used as a padding character. The term **`n1`** is the expected length of the output. Asterisks will be written if **`num`** requires more width than **`n1`**. Otherwise, the padding character **`left`** will be used to make up the remaining spaces. Pass a function (**`put`**) that will output one character. The function **`put`** should take a character argument. The term **`num`** is the number to be converted and output. This function can be called from processes of different priorities.

- **`void phex( char left, int n1, int(*put)(), int num )`**

  Similar to **`plhex`**. This function prints the hexadecimal representation of a short integer (signed or unsigned). Refer to the description of **`plhex`** for details.

- **`void pflt( char flag, char code, int width, int digits, int(*put)(), float value, int prec )`**

  Prints a formatted floating-point value using the specified single-character output procedure **`put`**. The programmer has quite a bit of control over format.

The **`flag`**, if **`'-'`**, indicates that the output field is left-justified. If it is **`'0'`**, the field is right-justified and zero-filled. Otherwise the field is right-justified and space-filled.

The **`code`** can be **`'e'`**, **`'f'`**, or **`'g'`**. These formats correspond to programming conventions established many years ago. E format displays a mantissa with "e" and an exponent. F format is standard decimal format. G format allows the compiler to decide whether to use "e" or "f" format.

The term **`width`** is the field width. If zero is specified, the field will be as wide as needed to represent **`value`**. The terms **`prec`** and **`digits`** govern the number of significant digits to print. If **`prec`** is non-zero (true), the function prints **`digits`** significant digits. Otherwise, the function prints six significant digits.

The function prints all asterisks if the value does not fit in the field specified.

- **`char *itoa( int value, char *buf )`**

  Converts signed integer **`value`** to a character string in **`*buf`**, with a minus sign in first place, when appropriate. The function suppresses leading zeros, but leaves one zero digit for **`value`** = 0. The maximum value is 32767. The function returns a pointer to the end (the null terminator) of the string in **`*buf`**.

- **`char *utoa( uint value, char *buf )`**

  Converts unsigned integer **`value`** to a character string in **`*buf`**. The function suppresses leading zeros, but leaves one zero digit for **`value`** = 0. The maximum value is 65535. The function returns a pointer to the end (the null terminator) of the string in **`*buf`**.

- **`char *htoa( int value, char *buf )`**

  Converts integer **`value`** to hex character string in **`*buf`**. Leading zeros are not suppressed. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

- **`char *hltoa( long value, char *buf )`**

  Converts long integer **`value`** to hex character string in **`*buf`**. Leading zeros are not suppressed. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

- **`char outchrs( char c, int n,int(*put)() )`**

  Uses single-character output function **`put`** to output **`n`** times the character **`c`** . The function **`put`** should take a character parameter.

  The function returns the value of character **`c`**.

- **`char *outstr( char *buf, int(*put)() )`**

  Outputs the string **`*buf`** using calls to single-character output function **`put`**. The function **`put`** should take a character parameter. The function returns a pointer to the end (null terminator) of the string in **`*buf`**.

STRING.LIB

The following are standard C string functions.

- **`float atof( char *sptr )`**

  Converts a character string to a floating-point value. The initial "white space" is ignored. This is ANSI compatible. The function returns the converted value.

- **`int atoi( char *sptr )`**

  Converts a character string to an integer value. The initial "white space" is ignored. This is ANSI compatible. The function returns the converted value.

- **`int atol( char *sptr )`**

  Converts a character string to a long integer value. The initial "white space" is ignored. This is ANSI compatible. The function returns the converted value.

- **`void *memset( void* dst, byte ch, uint n )`**

  Sets the memory starting at **`dst`** to **`n`** occurrences of the byte **`ch`**. The function returns a pointer to the address following the last byte written.

- **`char *strcpy( char *dst, char *src )`**

  Copies string **`*src`** to string **`*dst`**. The function copies at least one byte (the null). The function returns a pointer to **`*dst`**.

- **`char *strncpy( char *dst, char *src, uint n )`**

  Copies at most **`n`** characters from **`*src`** to **`*dst`**. May terminate earlier if null terminator is encountered in **`*src`** before **`n`** characters. The null terminator is not copied if **`n`** is encountered before null terminator (i.e., the programmer should take care of length-delimited cases). The function returns a pointer to **`*dst`**.

- **`char *strcat( char *dst, char *src )`**

  Concatenates string **`*src`** to the end of **`*dst`**. The destination string must be large enough to hold the additional characters. The function returns a pointer to **`*dst`**.

- **`char *strncat( char *dst, char *src, uint n )`**

  Concatenates up to **`n`** characters from **`*src`** to the end of **`*dst`**. A null terminator is appended to the end of **`*dst`** if **`n`** characters are copied before encountering the null terminator in **`*src`**. The function returns a pointer to **`*dst`**.

- **`int strcmp( char *a, char *b )`**

  Compares two strings. The function returns the relative difference between the first pair of differing characters, that is, the function result is

= 0	if all characters are equal
< 0	if $a_i < b_i$
> 0	if $a_i > b_i$.

  These functions are useful for sorting.

- **`int strncmp( char *a, char *b, uint n )`**

  Compares two strings up to **`n`** characters. The function return is similar to that of **`strcmp`**.

- **`char* strchr( char *src, char ch )`**

  Scans **`*src`** for the first occurrence of **`ch`**. The function returns a pointer to the first occurrence of **`ch`** in **`*src`**. It returns a null pointer if **`ch`** is not found.

- **`char* strrchr( char *src, int ch )`**

  Similar to **`strchr`**, except this function searches in reverse from the end of **`*src`** to the beginning. The function returns a pointer to the last occurrence of **`ch`** in **`*src`**. It returns a null pointer if **`ch`** is not found.

- **`uint strspn( char *src, char *set )`**

  Returns the length of the maximum initial segment of **`*src`**, which consists entirely of characters in **`*set`**.

- **`uint strcspn( char *src, char *set )`**

  Returns the length of the maximum initial segment of **`*src`**, which consists of characters not in **`*set`**.

- **`char* strpbrk( char *s1, char *s2 )`**

  Locates the first occurrence within **`*src`** of any character in **`*set`**. The function returns a pointer to the occurrence. The function returns a null pointer if none is found.

- **`void* memcpy( void *dst, void *src, uint n )`**

  Copies **`n`** characters from memory **`*src`** to memory **`*dst`**. Overlap is handled correctly. The function returns the **`*dst`** pointer.

- **`void* memchr( void* src, int ch, uint n )`**

  Searches up to **`n`** characters in buffer **`*src`** for character **`ch`**. The function returns a pointer to first occurrence of **`ch`** if found within **`n`** characters. Otherwise returns a null pointer.

- **`int strlen( char *s )`**

  Calculates the length of string **`*s`**, not including the terminating null. The function returns the number of bytes in the string.

- **`int toupper( int c )`**

  Converts character **`c`** to its upper-case equivalent.

- **`int tolower( int c )`**

  Converts character **`c`** to its lower-case equivalent.

- **`int islower( int c )`**

  Checks whether **`c`** is a lower-case character. The function returns non-zero if so, and zero otherwise.

- **`int isupper( int c )`**

  Checks whether **`c`** is an upper-case character. The function returns non-zero if so, and zero otherwise.

- **`int isdigit( int c )`**

  Checks whether **`c`** is an ASCII digit (0–9). The function returns non-zero if so, and zero otherwise.

- **`int isxdigit( int c )`**

  Checks whether **`c`** is a hexadecimal digit (0–9, a–f,. A–F). The function returns non-zero if so, and zero otherwise.

- **`int ispunct( int c )`**

  Checks whether **`c`** is a punctuation mark. The function returns non-zero if so, and zero otherwise.

- **`int isspace( int c )`**

  Checks whether **`c`** is a blank, tab, new line, or form feed. The function returns non-zero if so, and zero otherwise.

- **`int isprint( int c )`**

  Checks whether **`c`** is printable. The function returns non-zero if so, and zero otherwise.

- **`int isalpha( int c )`**

  Checks whether **`c`** is an ASCII letter. The function returns non-zero if so, and zero otherwise.

- **`int isalnum( int c )`**

  Checks whether **`c`** is alphanumeric (A to Z, a to z and 0 to 9). The function returns non-zero if so, and zero otherwise.

- **`int isgraph( int c )`**

  Checks whether **`c`** is a visible printing character. The function returns non-zero if so, and zero otherwise.

- **`int iscntrl( int c )`**

  Checks whether **`c`** is a control character (less than 20_H). The function returns non-zero if so, and zero otherwise.

- **`float strtod( char *s, char **tailptr )`**

  Converts a string to a floating-point value. The term **`*s`** is the string to convert, and **`**tailptr`** is a pointer to a pointer to a character. **`**tailptr`** is assigned the stopping point of conversion in **`*s`** (so continuation is possible at **`**tailptr`**). If no conversion takes place, **`**tailptr`** returns 0L. The initial "white space" is ignored. This function is ANSI compatible. The function returns the converted value.

- **`long strtol( char *s, char **tail, int base )`**

  Converts a string to a long integer value. The term **`*s`** is the string to convert, **`**tail`** is assigned the last position of the conversion, and **`base`** indicates the radix of conversion, which may be from 2 to 36. When **`base`** is 0, the function converts according to C syntax. For example, if the string starts with "**`0x`**," the function will interpret the string in hexadecimal format. The function skips the initial "white space." The function sets the tail pointer **`**tail`** to the character position at which the conversion failed or finished. The next conversion may resume at the location specified by **`**tail`**. If no conversion takes place, **`**tail`** returns **`0L`**. The initial "white space" is ignored. This function is ANSI compatible. The function returns the converted value.

  Be careful with the double pointer.

- **`char *strtok( char *src, char *brk )`**

  Scans **`*src`** for tokens separated by delimiter characters specified in **`*brk`**. The first call takes a non-null **`*src`**. Subsequent calls with a null pointer for **`*src`** continue to search for tokens in the string. The function returns a pointer to the first character of the token. If it also finds a terminating delimiter, it changes it to a null character so that the token is terminated. This function modifies the source string. The function returns a null pointer if it does not find a token.

- **`char *strstr( char *string, char *target )`**

  Returns a pointer to the first occurrence of substring **`*target`** in **`*string`**. The function returns a null pointer if **`*target`** is not found in **`*string`**. The function returns the pointer string if the target is null.

- **`int memcmp( void *a, void *b, uint n )`**

  Compares two memory spaces **`a`** and **`b`** and returns the relative difference between the first pair of differing bytes, if any. Thus, the function result is

= 0	if all bytes are equal
< 0	if $a_i < b_i$
> 0	if $a_i > b_i$.

  The function will stop comparing after **`n`** bytes.

SYS.LIB

These are miscellaneous support functions.

- **`int setjmp( jmp_buf env )`**

  Stores the PC (program counter), SP (stack pointer) and other information about the current state into **`env`**. The saved information can be restored by executing **`longjmp`**. A typical program appears below.

```
switch(setjmp(e)){
  case 0:  // first time
    fx();  // fx() may take a longjmp
    break; // if we get here, fx() was successful

// if we get here, fx() must have called longjmp
  case 1:
    do exception handling
    break;

// similar to case 1, but different exception code.
  case 2:
    ...
}
f(){
  g()
  ...
}           // Here, exception code 2 causes
g(){        // jump back to setjmp occurrence,
  ...       // but causes setjmp to return 2.
  longjmp(e,2);// Therefore, case 2 in the switch
}           // statement executes
```

The function returns zero when it is executed. After **`longjmp`** is executed, the program counter, stack pointer, etc., are restored to the state when **`setjmp`** was executed the first time. However, this time, **`setjmp`** returns whatever value is specified by the **`longjmp`** statement.

- **`void longjmp( jmp_buf env, int value )`**

  Restores the stack environment saved in **`env`**. The integer value passed to **`longjmp`** is returned as the function result of **`setjmp`** when the long jump is taken. See the description of **`setjmp`** for usage.

- **`void *malloc( uint size )`**

  Allocates a dynamic block of **`size`** bytes. Call **`bfree`** before using **`*malloc`** (the compiler automatically calls **`bfree`** before **`main`** if some heap space is reserved in the logical memory options). Because **`*malloc`** uses a global free list pointer, **`*malloc`** must be preempted by another **`*malloc`**. Heap space must be allocated using the logical memory option from the **Options** menu in order to use **`*malloc`**. (The default is a heap size of 0.) The function returns a pointer to the beginning of the allocated block, or a null pointer if space is unavailable.

- **`uint bfree( void *lo, void *hi )`**

  Defines a block of RAM, from **`*lo`** to **`*hi`** inclusive, as available for dynamic allocation. The function returns non-zero if successful, and zero if not.

- **`int free( void *f )`**

  Returns block (**`*f`**) of dynamically allocated RAM to the free list. The function returns non-zero if successful, and zero if not.

- **`int pack( void )`**

  Reduces fragmentation of dynamic memory by linking adjacent free blocks. The function returns the total number of free bytes.

- **`void *calloc( uint count, uint size )`**

  Allocates memory from the “heap” for a space of **`count`** elements of **`size`** bytes. The function finds a block of memory on the free list, trims it to the right size, and returns a pointer to the block. The function initializes the space to all zeros. The function returns a pointer to the allocated block, and returns a null pointer if it cannot find a block.

- **`void swap( byte a[], byte b[], int s )`**

  Swaps array **`a`** with array **`b`**, byte-for-byte, for the first **`size`** bytes.

- **`int qsort( void *base, uint n, uint s, int(*cmp)() )`**

  Performs a "quicksort" with center pivot, stack control, and easy-to-change comparison method. The term **`*base`** points to the base of an array (of fixed-size structures) to be sorted. The value **`n`** is the number of elements to be sorted, and **`s`** is the size of each element in the array. The programmer must supply a comparison function **`cmp`** that indicates the order of two structures. The comparison function take pointers to two structures

  ```
    int cmp( *p, *q )
  ```

  and returns –1 if the first is *less* than the second, 0 if the structures are equal, and 1 if the first is *greater* than the second one.

  The **`qsort`** function returns zero if the operation is successful, and non-zero otherwise.

- **`char *realloc( void *ptr, uint size )`**

  Allocates a new block of size **`size`**, copies the data from the old block (**`*ptr`**) to the new block, frees the old block, and returns a pointer to the new block. If the function fails to allocate a new block, the function result is a null pointer.

- **`isr_ptr getvect( uint intrno )`**

  Gets the address of the handler of interrupt number **`intrno`**. For this function, number must be even and less than 255. The function returns the address of the handler. The type **`isr_ptr`** is a pointer to a function that returns void and takes no arguments.

- **`void setvect( uint intrno, isr_ptr isr )`**

  Sets a new handler **`isr`** for interrupt number **`intrno`**. The term **`intrno`** must be even and less than 255. The type **`isr_ptr`** is a pointer to a function that returns void and takes no arguments.

- **`int iff()`**

  Checks whether the interrupt flag is on. The function returns 1 if the interrupt flag is on, and 0 otherwise.

- **`void setireg( char value )`**

  Sets the Z180 interrupt register with the upper 8 bits of the specified 16-bit **`value`**.

- **`char readireg()`**

  Returns the value of the Z180 interrupt register as the upper 8 bits of the returned value. The lower 8 bits are set to zero.

- **`void CoBegin ( CoData *cd )`**

  `CoBegin` initializes a **`CoData`** structure. The INIT flag is set, but the STOPPED flag is cleared.

- **`void CoReset ( CoData *cd )`**

  `CoReset` resets a **`CoData`** structure. The STOPPED and INIT flags are *both* set.

- **`void CoPause ( CoData *cd )`**

  `CoPause` pauses a **`CoData`** structure. The STOPPED flag is set, but the INIT flag is cleared.

- **`void CoResume ( CoData *cd )`**

  `CoResume` resumes a **`CoData`** structure. The STOPPED and INIT flags are both cleared.

- **`int isCoDone ( CoData *cd )`**

  The function **`isCoDone`** returns true (1) if both the STOPPED and INIT flags are set. It returns 0 otherwise.

- **`int isCoRunning ( CoData *cd )`**

  The function **`isCoRunning`** returns true (1) if the STOPPED flag is not set. It returns 0 otherwise.

- **`void _prot_init()`**

  Performs super initialization. The function initializes internal data needed for recovery of **`protected`** variables after a crash. To ensure that the protection mechanism works, call this function once in a program *before* any **`protected`** variables are set.

- **`void _prot_recover()`**

  Performs recovery of a partially completed assignment to a **`protected`** variable. Call this function after a power failure or a similar situation that does not lose memory.

- **`void reload_vec( int vector, int(*function)() )`**

  Loads an interrupt service routine to specified vector location at run time.

  PARAMETERS: **`vector`** is the interrupt vector to be served.

  `*function` is the address of the interrupt service routine.

  `reload_vec` writes to the flash memory when executed on a controller with a flash EPROM. Be careful not to have this function call write repeatedly to the same flash EPROM address since the flash EPROM has a maximum of about 10,000 writes.

XMEM.LIB

These are extended memory functions.

- **`ulong xmadr( void* address )`**

  Converts logical address **`address`** to a physical address according to the memory mapping registers. Uses BBR, CBR and CBAR to determine the physical address of any given logical address. The function returns the physical address.

- **`char xgetchar( long address )`**

  Gets a character whose address is specified by the physical **`address`** (20 bits). The function returns the character value.

- **`int xgetint( ulong address )`**

  Gets an integer whose address is specified by the physical **`address`** (20 bits). The function returns the integer value.

- **`ulong xgetlong( ulong address )`**

  Gets a long integer whose address is specified by the physical **`address`** (20 bits). The function returns an unsigned long integer value.

- **`float xgetfloat( ulong address )`**

  Gets a floating-point value whose address is specified by the physical **`address`** (20 bits). The function returns the floating-point value.

- **`void xputchar( long address, char value )`**

  Stores a character **`value`** at a physical **`address`** (20 bits).

- **`void xputint( long address, int value )`**

  Stores an integer **`value`** at a physical **`address`** (20 bits).

- **`void xputlong( long address, long value )`**

  Stores a long-**`value`** integer at a physical **`address`** (20 bits).

- **`void xputfloat( ulong address, float value )`**

  Stores a float **`value`** at a physical **`address`** (20 bits).

- **`void xmem2root( ulong src, void* dst, uint n )`**

  Copies a block of **`n`** bytes from extended memory **`src`** to root **`*dst`**. The address **`src`** is a physical **`address`** (20 bits).

- **`void root2xmem( void *src, ulong dst, uint n )`**

  Copies a block of **`n`** bytes from root memory **`*src`** to extended memory **`dst`**. The address **`dst`** is a physical **`address`** (20 bits).

- **`uint xstrlen( ulong address )`**

  Returns the length of the string at the extended memory **`address`**. The address is a physical **`address`** (20 bits).

- **`uint x-makadr( ulong address )`**

  Computes the logical address from a physical **`address`**. The function also sets CBR to new page number and returns the logical address in HL. The old CBR is saved in **`af'`** **(**alternative register pair A and F). ***Never*** call this function from **`xmem`** functions. Z-World also recommends that this function not be called from C functions since it is easy to forget that a C function may be placed in **`xmem`** automatically.

- **`ulong a32-24( ulong address )`**

  Converts the 20-bit physical **`address`** (in a 32-bit integer) to a segmented (24-bit) address. Segmented addresses have the following structure.

8-bit CBR	16-bit Z180 address

- **`ulong a24_32( ulong address )`**

  Converts the 24-bit segmented **`address`** into a 20-bit physical address (in a 32-bit integer). The segment (second byte of the segmented address) is only effective if **`address`** is in **`xmem`**, that is, **`address`** ≥ 0xE000. Otherwise, the segment is ignored. Both the CBAR and BBR registers in the MMU are used to calculate the outcome. The function returns an unsigned long integer that holds the 20-bit physical address equivalent to the extended logical address supplied.

2

MULTITASKING LIBRARIES

The multitasking libraries described in Chapter 2 include the real-time kernel, the simplified real-time kernel, and the virtual driver.

RTK.LIB

This library is the full real-time kernel. The simplified real-time kernel (SRTK) is described later.

- **`int request( uint tasknum )`**

  Requests the kernel to run the task specified by **`tasknum`** immediately. If a request for the task is pending, this call has no further effect. The specified task will be run on a future tick when priorities allow.

- **`void run_every( int tasknum, int period )`**

  Requests the kernel to run the task specified by **`tasknum`** every **`period`** ticks. The first request comes after **`period`** ticks. This is exact and no ticks will be gained or lost in the period.

- **`void run_after( int tasknum, long delay )`**

  Requests the kernel to run the task specified by **`tasknum`** after **`delay`** ticks have occurred.

- **`void run_at( int tasknum, void* time )`**

  Requests the kernel to run the task specified by **`tasknum`** when the time is greater than or equal to the time specified by the pointer **`time`**. The time pointer points to a 48-bit number (stored least significant byte first) that is the number of ticks since **`init_kernel`** was called.

- **`void run_cancel( int tasknum )`**

  Cancels any pending requests for the task specified by **`tasknum`**.

- **`void gettimer( void* time )`**

  Returns the current 48-bit time to the 6-byte area to which **`time`** points.

- **`void run_timer()`**

  This function must be called by an interrupt routine between 10 and 500 times per second for the real-time kernel to operate. Each call to this function constitutes one kernel "tick," so all time values used by other kernel functions depend on the rate at which this function is called.

- **`int comp48( void* time1, void* time2 )`**

  Compares two 48-bit time values. The function returns

  –1 for **`time1`** < **`time2`**,
  0 for **`time1`** = **`time2`**, and
  +1 for **`time1`** > **`time2`**.

- **`void rkernel()`**

  This is the real-time kernel core, and is called by **`run_timer`**. This function will return immediately if there is no change to the task currently executing. If it decides to change tasks based on service requests such as **`run_every`** or **`run_after`**, then it will not return until the new task either returns or calls **`suspend`**.

- **`void suspend( uint ticks )`**

  This routine must be called only from within a given task. It allows the task to suspend itself for the specified number of ticks, after which it will continue to be requested automatically. Execution resumes at the statement following the call to **`suspend`**.

  If **`ticks`** is 0, then the suspension is for an indefinite period of time, until the task is again requested by some outside agent, such as a call to **`run_every()`**. Using a **`while`** statement is the usual method of using **`suspend`** to wait for an external event:

  ```
  while( !event() ) suspend(20);
  ```

  This example checks for the event every 20 ticks until the event takes place, at which point execution continues. The suspension can be up to 65,535 ticks.

- **`int init_kernel()`**

  Initializes the real-time kernel. This function takes no parameters. However, the calling program must contain certain definitions.

  Functions to be run as tasks must be declared with no parameters and return an integer. A global array of task pointers, **`Ftask`**, must be declared with the first task (**`Ftask[0]`**) given the highest priority and the last task the lowest priority. **`#define NTASKS`** to be the number of tasks. Then set up a periodic interrupt with a service routine that calls **`run-timer`**. An option is to define **`TASKSIZE_STORE`** to be the size of the task storage area (this defaults to 50 if **`TASKSIZE_STORE`** is not defined).

All of the above definitions must occur in the source code before any reference to real-time kernel functions.

SRTK.LIB

These are the simplified real-time kernel functions.

- **`void srtk_hightask()`**

  This is the routine called every 25 ms by the SRTK to run high-priority tasks. The one in the library is a dummy routine.

  To have a user-defined SRTK high-priority task, simply write one with the same name. Specify **`#nointerleave`** to guarantee that the user-defined high-priority task is compiled.

- **`void srtk_lowtask()`**

  This routine is called every 100 milliseconds by the SRTK to run low-priority tasks. The one in the library is a dummy routine.

  To have a user-defined SRTK high priority task, simply write one with the same name. Specify **`#nointerleave`** to guarantee that the user-defined high-priorirty task is compiled.

- **`void init_srtkernel()`**

  Initializes the simplified real-time kernel. Once this is called, periodic interrupts will automatically invoke the SRTK high- and low-priority tasks.

  Initialize the virtual driver and **`#define RUNKERNEL 1`** before calling this function.

VDRIVER.LIB

These are the virtual driver functions. The virtual driver provides a number of different services, such as the virtual watchdog timers and a “fastcall” very high priority task.

The virtual driver also provides delay routines for use by **`waitfor`** statements **`DelayMs`**, **`DelaySec`**, and **`DelayTick`**.

- **`void VdInit()`**

  Initializes the virtual driver. The Z180 PRT1 clocks the virtual driver every 1/1280 second. The virtual driver clocks the RTK or SRTK every 32 ticks (or 25 milliseconds) if **`#define RUNKERNEL`** is defined.

  For fastcall service, the virtual driver clocks **`vd–quickloop`** every **`n`** ticks (1/1280 seconds) where $1 \leq n \leq 255$. **`vd–quickloop`** must be defined and the definition will override the dummy version in the library. (**`#define VD–FASTCALL 1`** must be defined as well.)

  `VdInit` must be called before the program can use the SRTK, virtual watchdogs, the **`waitfor`** delay routines or fastcall.

VdInit makes a call to **_GLOBAL_INIT**. Therefore, a user-prepared program does not have to.

- **int VdGetFreeWd(byte count)**

  Returns a free virtual watchdog timer and starts it counting down from **count**. Virtual watchdog timers decrement every 25 milliseconds (32 virtual driver ticks). When a virtual watchdog reaches 0, it resets the processor. Once a virtual watchdog timer is active, the software should reset the timer periodically with a call to **VdWdogHit**. The function returns the integer ID of an unused virtual watchdog timer.

  If **count** ≤ 2, **VdWdogHit** must be called every 25 milliseconds. If **count** = 255, hit the watchdog at least every 6.375 seconds.

- **void VdWdogHit(int wd)**

  Resets virtual watchdog timer to *n* counts where *n* was the argument to the call to **VdGetFreeWd** that obtained the virtual watchdog **wd**. The function returns 0 if **wd** is out of range, and 1 if successful.

- **int VdReleaseWd(int wd)**

  Deactivates a virtual watchdog **wd** and returns it to the pool of watch-dogs. The function returns 0 if **wd** is out of range, and 1 if successful.

- **int vd_initquickloop(int n)**

  Initializes the "fastcall" feature of the virtual driver to run every **n** ticks. The value of **n** must be from 0 to 255. If **n** = 0, it turns off **fastcall**. Use **#define VD_FASTCALL 1**, call **VdInit**, then call this function. (**VdInit** initializes fastcall as *off*.) The function returns 1 for success, 0 for a bad **n** value.

- **void VdAdjClk()**

  Synchronizes the software second timer used by **DelaySec** with the real-time clock. Call this function once a day or so to keep clocks in sync.

- **vd_fastcall()**

  Is called by the virtual driver to run an ultra-fast thread every **n** ticks, where **n** is the argument to **vd_initquickloop()** and should be between 0 and 255. Use **#define VD_FASTCALL 1** to activate this thread. **n** = 0 shuts off **fastcall**.

3

CONTROLLER LIBRARIES

Each of the libraries described in Chapter 3 is the principal library for one type of Z-World controller. Some of the libraries in Chapter 5 also support particular controllers.

BL1000.LIB

This function supports the BL1000 controller.

- **`int ad_rd8( int chan )`**

  Reads an 8-bit value from the BL1000 A/D converter. **`chan`** is the channel number (0–3). The return value is shifted left by 4 bits, so it appears as a 12-bit number.

  The function returns 0 to 4095 if successful, –32768 if an error occurred.

BL11XX.LIB

These functions support the BL1100 series controllers.

- **`int ad_rd10( int chan )`**

  Reads a 10-bit value from the BL1100 A/D converter. The low 3 bits of **`chan`** specify the channel number (0–7); the fourth bit must be 0 for bipolar mode, or 1 for unipolar mode (add 8 to the channel number for unipolar mode). The return value is shifted left by 2 bits, so it appears as a 12-bit number.

  The function returns –2048 to 2047 if bipolar mode; 0 to 4095 if unipolar mode; –32768 if an error occurs.

- **`int ad_rd12( int chan )`**

  Reads a 12-bit value from the BL1100 A/D converter. The low 3 bits of `chan` specify the channel number (0–7); the fourth bit must be 0 for bipolar mode, or 1 for unipolar mode (add 8 to the channel number for unipolar mode).

  The function returns –2048 to 2047 if bipolar mode; 0 to 4095 if unipolar mode; –32768 if an error occurs.

- **`int ad_rd10s( int chan, int count, int *buf, uint divider )`**

  Samples data from the BL1100 A/D converter at uniform intervals in time. **`chan`** is the channel number (0–7), plus 8 for unipolar mode (otherwise bipolar), **`count`** specifies the number of samples to collect, and **`buf`** points to a buffer where the samples will be stored. **`divider`** determines the sample rate based on the formula rate = clock/(20***`divider`**). **`divider`** should not be smaller than 36, which yields 12800 samples per second with a 9.216 MHz clock. Interrupts will be disabled unless **`NODISINT`** is defined.

  The function returns 1 if successful, 0 if the sample was missed because **`divider`** is too small or there was an interrupt during sampling.

- **`int ad_rd12a( int chan )`**

  Reads a 12-bit value from the BL1100 alternate A/D converter LTC1290. The low 3 bits of **`chan`** specify the channel number (0–7); the fourth bit must be 0 for bipolar mode, or 1 for unipolar mode (add 8 to the channel number for unipolar mode). The execution time is about 350 microseconds with a 9.216-MHz system clock. Interrupts are disabled for about 300 microseconds.

- **`void wdac( int value )`**

  Writes **`value`** to the BL1100 DAC. **`value`** should be in the range 0–4095, with an output of 2.5***`value`**/4096 volts.

- **`int ad_rd( int chan )`**

  Same as **`ad_rd10`**.

- **`void setctc( char nctc, char mode, char timer, char intr )`**

  Initializes CTC counter **`nctc`** (0–3). **`mode`** specifies one of seven possible counter modes as follows.

0	runs the counter at sysclock/16, triggering immediately.
1	runs the counter at sysclock/256, triggering immediately.
2	sets the counter to run off of an external clock.
4	runs the counter at sysclock/16, triggering on the rising edge of CLK/.
5	runs the counter at sysclock/256, triggering on the rising edge of CLK/.
6	runs the counter at sysclock/16, triggering on the falling edge of CLK/.
7	runs the counter at sysclock/256, triggering on the falling edge of CLK/.

  `timer` specifies the time constant to load into the counter. **`intr`** indicates whether or not the timer should cause interrupts (non-zero enables interrupts, zero disables interrupts).

  The function returns void.

- **`void setdaisy( char code )`**

  Sets the relative priority of interrupts between the three I/O units in the KIO based on the value of **`code`** shown below.

0	disabled	4	CTC,PIO,SIO
1	SIO,CTC,PIO	5	PIO,SIO,CTC
2	SIO,PIO,CTC	6	PIO,CTC,SIO
3	CTC,SIO,PIO (system default)	7	disabled

- **`void setled1( char value )`**

  Turns LED #1 on if **`value`** is non-zero, off if **`value`** is zero.

BL14_15.LIB

These functions support the BL1400 series controllers.

- **`int Read555( uint *lapsecount )`**

  Reads timer0 count for the amount of time it took the 555 chip to reach Δt = 1.1RC time. The timer count is returned in **`*lapsecount`**. The 555 chip should be set previously with **`Set555(maxcount)`**.

  The function returns

  0 if timer0 has not timed out and the 555 chip has not reached t = 1.1RC time.

  1 if the 555 has reached t = 1.1RC time and has generated an interrupt on INT1 and DREQ0.

  –1 if timer0 has finished counting **`maxcount`** and the 555 has not reached t = 1.1RC time.

- **`void Set555( uint maxcount )`**

  Loads timer0 with **`maxcount`** and sets it to generate one interrupt. Prepares DMA0 to receive data from timer0 TMRD0L. Prepares INT1 and DREQ0 to receive a done signal from the 555 chip. Triggers the 555 chip.

- **`void Charger1302( int on–off, int diode, int resistor )`**

  Turns the trickle charger on the DS1302 chip on or off. **`diode`** is 1 or 2 for the number of diodes from VCC2 to VCC1. **`resistor`** is 2, 4 or 8 for the resistance (in kΩ) across the line.

- **`int ReadTime1302( struct tm* thistime )`**

  Reads real-time clock (RTC) data from DS1302 to the time structure pointed to by **`thistime`**. The function returns 0 if successful, –1 if the RTC is in halt mode.

- **`int WriteTime1302( struct tm* thistime )`**

  Writes time structure data pointed to by **`thistime`** to the real-time clock (RTC) of the DS1302. The function returns 0 if successful, –1 if the RTC is in halt mode.

- **`void WriteRam1302( int ram_loc, int data )`**

  Writes data to **`ram_loc`** (0–30) of the DS1302. The function returns 1 if the write is successful, and –1 if an error occurs.

- **`int ReadRam1302( int ram_loc )`**

  Reads data from **`ram_loc`** (0–30) of the DS1302. The function returns RAM data, or –1 if an error occurs.

- **`void WriteBurst1302( char *pdata, int count )`**

  Writes **`count`** bytes from the array **`pdata`** to the RAM of the DS1302 ,starting at RAM location 0.

- **`void ReadBurst1302( char *pdata, int count )`**

  Reads back **`count`** number of bytes from the DS1302, starting from RAM location 0 to the array **`pdata`**.

- **`void Write1302( int reg, int data )`**

  Writes **`data`** to the specified register of the DS1302.

- **`int Read1302( int reg )`**

  Reads data from the specified register of the DS1302. The function returns the data read.

- **`void setPIOCA( byte mask )`**

  Active bits (1s) of **`mask`** are set in **`PIOCAShadow`**. That result is then sent to **`PIOCA`**. Active bits become *input* bits.

  `PIOCA` ← **`PIOCAShadow`** ← **`PIOCAShadow`** OR **`mask`**

- **`void resPIOCA( byte mask )`**

  Active bits (1s) of **`mask`** are reset in **`PIOCAShadow`**. That result is then sent to **`PIOCA`**. Active bits become *output* bits.

  `PIOCA` ← **`PIOCAShadow`** ← **`PIOCAShadow`** AND NOT **`mask`**

- **`void setPIODA( byte mask )`**

  Active bits (1s) of **`mask`** are set in the current output of **`PIODA`**.

  `PIODA` ← **`PIODA`** OR **`mask`**

- **`void resPIODA( byte mask )`**

  Active bits (1s) of **`mask`** are reset in the current output of **`PIODA`**.

  `PIODA` ← **`PIODA`** AND NOT **`mask`**

- **`void setPIOCB( byte mask )`**

  Active bits (1s) of **`mask`** are set in **`PIOCBShadow`**. That result is then sent to **`PIOCB`**. Active bits become *input* bits.

  `PIOCB` ← **`PIOCBShadow`** ← **`PIOCBShadow`** OR **`mask`**

- **`void resPIOCB( byte mask )`**

  Active bits (1s) of **`mask`** are reset in **`PIOCBShadow`**. That result is then sent to **`PIOCB`**. Active bits become *output* bits.

  `PIOCB` ← **`PIOCBShadow`** ← **`PIOCBShadow`** AND NOT **`mask`**

- **`void setPIODB( byte mask )`**

  Active bits (1s) of **`mask`** are set in the current output of **`PIODB`**.

  `PIODB` ← **`PIODB`** OR **`mask`**

- **`void resPIODB( byte mask )`**

  Active bits (1s) of **`mask`** are reset in the current output of **`PIODB`**.

  `PIODB` ← **`PIODB`** AND NOT **`mask`**

- **`void mgset12adr( int addr )`**

  Sets the current address of the PLCBus. A subsequent read or write of the PLCBus will access the expansion device with this address. The address remains in effect until a new address is set. The term **`addr`** is the 12-bit physical address of the PLCBus device. The lowest 4-bit nibble is transmitted last (as **`BUSADR2`**). The third nibble is transmitted first (as **`BUSADR0`**).

- **`void mgwrite12data( int addr, int data )`**

  Writes data to the PLCBus device at **`addr`**. Only the lowest four bits of data are useable (for **`BUSWR`**).

- **`int mgread12data0( int addr )`**

  Reads data (with **`BUSRD0`**) from the PLCBus device at `addr`. The function result holds the data.

- **`int mgread12data1( int addr )`**

  Reads data (with **`BUSRD1`**) from the PLCBus device at **`addr`**. The function result holds the data.

- **`int mgread12data2( int addr )`**

  Reads data (with **`BUSRD2`**) from the PLCBus device at **`addr`**. The function result holds the data.

- **`void mgwrite4data( int value )`**

  Writes the low 4 bits of value (with **`BUSWR`**) to a PLCBus device. This function assumes that the PLCBus device's address has been placed on the bus (with **`mgset12adr`**).

- **`void mgsave_pbus()`**

  Saves the current state of the PLCBus to the stack. This function should only be called in tandem with **`mgrestore_pbus`**. Otherwise, the stack will become unbalanced.

- **`void mgrestore_pbus()`**

  Restores the current state of the PLCBus from the stack. This function should only be called in tandem with **`mgsave_pbus`**. Otherwise, the stack will become unbalanced.

- **`void mgplc_set_relay( int number, int relay, int state )`**

  Turns a relay on the PLCBus on or off. The board must be a Z-World XP8300 or XP8400 board and its **`number`** must be from 0 to 63. The term **`relay`** selects the relay on the selected board (0–5 for XP8300 boards and 0–7 for XP8400 boards). The term **`state`** is 1 to turn the relay on and 0 to turn it off.

  Refer to the ***XP8300, XP8400 and SE1100 User's Manual*** for details regarding devices and device numbering schemes.

- **`int mgplcrly_board( int number )`**

  Computes the physical address of a relay board from its board **`number`**. The number must be from 0 to 63. (Board number 0 corresponds to address 0x000; board number 63 corresponds to address 0x11F.) The return value has the third and the first nibbles interchanged.

  Refer to the ***XP8300, XP8400 and SE1100 User's Manual*** for details regarding devices and device numbering schemes.

- **`int mgplcuio_board( int number )`**

  Computes the physical address of a universal I/O board (XP8700) from its board **`number`**. The number must be from 0 to 15. (Board number 0 corresponds to address 0x040; board number 15 corresponds to address 0x04F.) The return value has the third and the first nibbles interchanged.

  Refer to the ***XP8100 and XP8200 User's Manual*** for details regarding devices and device numbering schemes.

- **`int mgplc_dac_board( int number )`**

  Computes the physical address of a DAC board (XP8600) from its board **`number`**. The number must be from 0 to 63. (Board number 0 corresponds to address 0x020 board number 63 corresponds to address 0x13F.) The return value has the third and the first nibbles interchanged.

  Refer to the ***XP8600 and XP8900 User's Manual*** for details regarding devices and device numbering schemes.

- **`void mginit_dac()`**

  Initializes a DAC board (XP8600) board on the PLCBus. This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**).

- **`void mgwrite_dac1( int value )`**

  Writes the 12-bit integer **`value`** to Register A of DAC 1 of a DAC board (XP8600)on the PLCBus. This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). The DAC board does not produce a new conversion value until a call to **`mglatch_dac1`** is executed.

- **`void mglatch_dac1()`**

  Moves Register A data to Register B of DAC 1 of a DAC board (XP8600) on the PLCBus. Actual DAC 1 output is converted from Register B. This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). Ensure that Register A contains valid data. See **`mgwrite_dac1`** above.

- **`void mgset_dac1( int value )`**

  Writes a 12-bit integer value to Register A, then moves the data from Register A to Register B of DAC 1 of a selected DAC board (XP8600). This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). It combines the effect of **`mgwrite_dac1`** and **`mglatch_dac1`**.

- **`void mgwrite_dac2( int value )`**

  Writes the 12-bit integer value to Register A of DAC 2 of a DAC board (XP8600) on the PLCBus. This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). The DAC board does not produce a new conversion value until a call to **`mglatch_dac2`** is executed.

- **`void mglatch_dac2()`**

  Moves Register A data to Register B of DAC 2 of a DAC board (XP8600) on the PLCBus. Actual DAC 2 output is converted from Register B. This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). Ensure that Register A contains valid data. See **`mgwrite_dac2`** above.

- **`void mgset_dac2( int value )`**

  Writes a 12-bit integer value to Register A, then moves the data from Register A to Register B of DAC 2 of a selected DAC board (XP8600). This function assumes that the board's address has been placed on the bus (with **`mgset12adr`**). It combines the effect of **`mgwrite_dac2`** and **`mglatch_dac2`**.

- **`void lc_char( byte data )`**

  Writes a character to the LCD.

- **`void lc_ctrl( byte cmd )`**

  Writes a control command to the LCD.

- **`void lc_init()`**

  Initializes the LCD and accessory variables. The LCD uses PIO Port A of the BL1400.

- **`void lc_cgram( void* ptr )`**

  Loads up to 8 special characters to the character generator of the LCD from the byte array **`*p`**. The first byte in the array is the number of bytes to store (at 8 bytes per character), with a maximum value of 64 for 8 characters. The character codes for the special characters are 0, 1, 2, 3, 4, 5, 6, and 7.

- **`void lc_printf( char* format, ... )`**

  This function works like **`printf`**, but for the LCD. The following escape sequences are also implemented:

ESC 1	turns cursor on
ESC 0	turns cursor off
ESC c	erases from cursor to end of line
ESC b	enables cursor blinking
ESC n	disables cursor blinking
ESC e	erases display and home cursor
ESC p n mm	position cursor at line n, column mm

The escape character code is **`0x1B`**.

- **`lc_kxinit()`**

  Initializes the keypad driver and accessory variables. Be sure to define **`KEY4x6`** somewhere at the start of the code for a 4 × 6 keypad.

  ```
  #define KEY4x6
  ```

  Otherwise, the driver defaults to a 2 × 6 keypad.

- **`int lc_kxget( int mode )`**

  Obtains the key value from the FIFO keypad buffer. If **`mode`** = 0, the key value is removed from the buffer. Otherwise, the key value is left in the buffer. In either case, the function returns the key value, or –1 if the keypad buffer is empty.

- **`void lc_keyscan()`**

  Scans the 4 × 6 or 2 × 6 keypad. A valid key has to be persistent for **`DebounceCount`** calls to **`lc_keyscan`**. The function puts valid keypresses into the keypad FIFO buffer. The software will access these keypresses using **`lc_kxget`**.

  "Debouncing" is done by making sure a key is pressed for **`DebounceCount`** consecutive calls to **`lc_keyscan`**. The debouncing number may be changed by redefining **`DebounceCount`**:

  ```
  #define DebounceCount nn
  ```

  If not redefined, **`DebounceCount`** defaults to 20. If **`lc_keyscan`** is called every 25 milliseconds and **`DebounceCount`** is 20, then a key has to be pressed for 20 × 25 milliseconds = 500 milliseconds to be valid.

BL16XX.LIB

These functions support the BL1600 series controllers.

- **`void VIOInit()`**

  Dummy function used as a host for global initialization of the virtual I/O variables. Virtual inputs are read and virtual outputs are written out whenever the function **`VIODrvr`** is called. Digital inputs are **`DIGIN1`** to **`DIGIN12`**. Digital outputs are **`OUTB1`** to **`OUTB8`** and **`HC1`** to **`HC6`**. A digital input must have the same value for two successive reads to be valid.

- **`void VIODrvr()`**

  Updates the virtual inputs **`DIGIN1`** through **`DIGIN12`**. The virtual outputs **`OUTB1`** to **`OUTB8`** and **`HC1`** to **`HC6`** are sent out to corresponding output ports.

- **`int up_digin( int channel )`**

  Reads the value of a digital input channel, **`channel`** must be from 1 to 12. The function returns 1 or 0, depending on the state of the channel.

- **`void up_setout( int channel, int onoff )`**

  Sets a digital output to 1 (active) or 0 (inactive), **`channel`** must be from 1 to 14. Channels 1–8 correspond to **`OUTB1`** through **`OUTB8`**, channels 9–14 correspond to **`HC1`** through **`HC6`**. The term **`onoff`** is the output value for the channel: 1 => high or active, 0 => low or inactive.

PK21XX.LIB

These functions support the PK2100 series controllers.

- **`void VIOInit()`**

  Dummy function used as a host for global initialization of the virtual I/O variables. Virtual inputs are read and virtual outputs are written out whenever the function **`VIODrvr`** is called. Digital inputs are **`DIGIN1`** to **`DIGIN7`** and **`U1IN`** to **`U7IN`**. Digital outputs are **`OUT1`** to **`OUT10`**, **`RELAY1`** and **`RELAY2`**. A digital input must have the same value for two successive reads to be valid.

- **`void VIODrvr()`**

  Updates the virtual inputs **`DIGIN1 to DIGIN7`** and **`U1IN`** to **`U7IN`**. The virtual outputs **`OUT1`** to **`OUT10`**, **`RELAY1`** and **`RELAY2`** are sent out to corresponding output ports.

- **`int up_digin( int channel )`**

  Reads value of digital input channel. **`channel`** must be from 1 to 7. The function returns 1 or 0, depending on the state of the channel.

- **`void up_setout( int nout, int onoff )`**

  Sets a digital output to 1 (active) or 0 (inactive). **`channel`** must be from 1 to 10, corresponding to **`OUT1`** to **`OUT10`**, 11 for **`RELAY1`**, or 12 for **`RELAY2`**. The term **`onoff`** is the output value for the channel: 1 => high or active; 0 => low or inactive.

- **`void init_daccal()`**

  Dummy function used as a host for global initialization of the DACCAL calibration values for the PK2100's DAC output.

- **`void up_daccal( int val )`**

  Outputs to DAC with calibration value for 0-10000 millivolts.

- **`void up_dacout( int val )`**

  Outputs uncalibrated D/A value to channel DAC.

- **`void up_expout( int val )`**

  Outputs uncalibrated D/A value to channel EXP.

- **`int up_adtest( int chan, int testval )`**

  Compares voltage input at universal input channel **`chan`** to **`testval`**. The channel must be 1–7. The function returns 1 if the input voltage is greater than **`testval`**, otherwise it returns 0.

- **`int up_uncal( int val )`**

  Returns uncalibrated integer (0-1023), given calibrated D/A output value in millivolts. The function returns the integer equivalent of the input value in millivolts.

- **`int up_docal( int rawval )`**

  Converts **`rawval`** to its calibrated value as A/D input.

- **`int up_adcal( int chan )`**

  Reads specified universal input channel (1–7). The function returns the calibrated A/D value for 0-10000 millivolts.

- **`int up_adrd( int chan )`**

  Reads universal input channel (1–7). The function returns the raw value (ADC output) from specified the channel.

- **`void up_dac420( int current )`**

  Outputs 4-20 milliamps at D/A channel DAC. Hardware must be configured for current-loop operation. The range for current is 4000–20000.

- **`int up_in420()`**

  Reads universal input channel 6 as a 4-20 milliamp input. Hardware must be configured for current-loop operation. The function returns a value in the range 4000–20000.

- **`float up_higain( int mode )`**

  Reads high-gain channel with H7 not jumpered. The function returns are as follows.

  If **`mode`** = 1, returns AD+ (0–1 volts). Assumes AD– is grounded.
  If **`mode`** = 2, returns AD+ – AD– (0–1 volts).
  If **`mode`** = 3, returns AD– (0–10 volts).
  If **`mode`** is undefined, returns –100.

PK22XX.LIB

These functions support the PK2200 series controllers.

- **`void VIOInit()`**

  Dummy function used as a host for global initialization of the virtual I/O variables. Virtual inputs are read and virtual outputs are written out whenever the function **`VIODrvr`** is called. Digital inputs are **`DIGIN1`** to **`DIGIN16`**. Digital outputs are **`OUT1`** to **`OUT14`**. A digital input must have the same value for two successive reads to be valid.

- **`void VIODrvr()`**

  Updates the virtual inputs **`DIGIN1`** through **`DIGIN16`**. The virtual outputs **`OUT1`** to **`OUT14`** are sent out to corresponding output ports.

- **`int up_digin( int channel )`**

  Reads value of digital input channel. The channel must be from 1 to 16. The function returns 1 or 0, depending on the state of the channel.

- **`void up–setout( int nout, int onoff )`**

  Sets a digital output to 1 (active) or 0 (inactive). The channel must be from 1 to 14, corresponding to **`OUT1`** through **`OUT14`**. The term **`onoff`** is the output value for the channel: 1 => high or active, 0 => low or inactive.

CM71_72.LIB

The CM71_72 library contains functions written specifically for the CM7100 and CM7200 series microprocessor core modules used in conjunction with their evaluation board and LCD/keypad module. Functions of the same name may exist in other libraries.

- **`void lc–kxinit()`**

  Initializes the keypad driver and accessory variables for a 2 × 6 keypad.

- **`int lc_kxget( int mode )`**

  Obtains the key value from the FIFO keypad buffer. If **`mode`** = 0, the key value is removed from the buffer. Otherwise, the key value is left in the buffer. In either case, the function returns the key value, or -1 if the keypad buffer is empty.

- **`void lc_keyscan()`**

  Scans the 2 × 6 keypad on the CM7100 evaluation board. A valid keypress has to persist for **`DebounceCount`** samples. (**`DebounceCount`** is defined globally with a default value of 10.) Call **`lc_keyscan`** in a periodic routine. Valid keys are placed in a keypad buffer. Access the keypad buffer with function **`lc_kxget`**.

- **`void up_beep( uint k )`**

  Starts the beeper sounding for k milliseconds. The number of milliseconds that the beeper actually sounds depends on the periodic routine that calls **`lc_beepscan`**. For example, if **`lc_beepscan`** is called every 50 milliseconds, then **`BeepScale`** = 1/50 = 0.02. (**`BeepScale`** is a globally defined value whose default is 0.04.)

- **`void lc_beepscan()`**

  Services the beeper on the CM7100 Evaluation Board. The beep duration is set previously with **`up_beep`**. The beeper count is decremented whenever this function is called. The beeper is turned off when the count reaches zero.

  This function should be called by a periodic routine, for example, one that executes every 25 milliseconds.

4 AASC Libraries

The Abstract Application-Level Serial Communication (AASC) library and its low-level support functions facilitate serial communication between controllers, and between a controller and another device such as a PC.

AASC.LIB

AASC libraries allow the programmer to create buffered character streams that perform input/output to/from ports in the communication devices. One principal library, **AASC.LIB**, contains all the functions required for these tasks.

The high-level routines handle the bookkeeping for the connections between the low-level circular buffer and hardware driver libraries. This allows the same programming framework to be used by any applicable hardware drivers.

- **CHANNEL aascOpen(int Type, char CRTS,
      long Param, void(*brqfnc)())**

  Opens a channel of device **Type**, and initializes the device with parameter **Param**.

  PARAMETERS: **Type** is the type of communication device to open.

  DEV_Z0 for the Z0 port,
  DEV_Z1 for the Z1 port,
  DEV_SCC for the Serial Communication Controller port,
  DEV_ZNET for the network device, and
  DEV_UART for the XP8700.

  CRTS specifies whether CTS/RTS handshaking should be used: 1 means CTS/RTS handshaking is used, 0 means CTS/RTS handshaking is *not* used.

  Param specifies all the other communication options. Z-World has defined the following macros.

Number of Data Bits	Number of Stop Bits	Number of Parity Bits
ASCI_PARAM_7D	**ASCI_PARAM_1STOP**	**ASCI_PARAM_NOPARITY**
ASCI_PARAM_8D	**ASCI_PARAM_2STOP**	**ASCI_PARAM_OPARITY**
		ASCI_PARAM_EPARITY
SCC_7DATA	**SCC_1STOP**	**SCC_NOPARITY**
SCC_8DATA	**SCC_2STOP**	**SCC_OPARITY**
		SCC_EPARITY

These macros apply to port Z0 of the Z180 or to the Serial Communication controller. Refer to the Dynamic C driver descriptions or online help for additional macros.

Choose one macro from each column to bit-or or add together to describe the channel configuration, as shown below.

```
ASCI_PARAM_7D | ASCI_PARAM_1STOP | ASCI_NOPARITY
```

Two commonly used combination macros have also been defined.

`ASCI_PARAM_1200`—Basic quantum for baud rate. Multiply by the factor baud rate ÷ 1200 (for example, 8 for 9600 bps).

`ASCI_PARAM_8N1`—Specifies 8 data bits, 1 stop bit and no parity.

For example, the Z0 channel in 8N1 format at 19,200 bps would have

```
Param = 16*ASCI_PARAM_1200 | ASCI_PARAM_8N1 .
```

`brqfnc` is a pointer to a function to be called by the **`Z0`** interrupt when a break request is detected. The return for **`void *brqfnc`** is null.

RETURN VALUE: 16-bit quantity of type **`CHANNEL`** for all further channel operations. **`aascOpen`** returns null if no channels can be assigned if break processing is not used.

- **`void aascClose( CHANNEL Channel )`**

  Closes the channel numbered **`Channel`**. First, **`aascClose`** calls the device-dependent routine to close the device. Then the storage associated with this channel is reattached to the free list.

  PARAMETER: **`Channel`** is the logical channel.

- **`void aascSetReadBuf( CHANNEL channel,`**
  `char *Buffer, uint size )`

  Designates a memory area pointed to by **`Buffer`** of **`size`** to be the receive buffer for **`channel`**.

  PARAMETERS: **`channel`** to be read from must be opened by an **`aascOpen`** call.

  `Buffer` is the address of the receive buffer.

  `size` is the size of the receive buffer.

- **`void aascSetWriteBuf( CHANNEL Channel,`**
  `char *Buffer, uint size )`

  Designates a memory area pointed to by **`Buffer`** of **`size`** to be the transmit buffer for **`Channel`**.

  PARAMETERS: **`Channel`** to write to must be opened by an **`aascOpen`** call.

  `Buffer` is the address of the transmit buffer.

  `size` is the size of the transmit buffer.

- **`void aascRxSwitch( CHANNEL Channel, char OnOff )`**

  Activates or deactivates the channel receiver.

  PARAMETERS: `Channel` is the logical channel.

  `OnOff` 0 is off, otherwise the channel transmitter is on.

- **`void aascTxSwitch( CHANNEL Channel, char OnOff )`**

  Switches the channel transmitter on or off.

  PARAMETERS: `Channel` is the logical channel.

  `OnOff` 0 is off, otherwise the channel transmitter is on.

- **`uint aascReadChar( CHANNEL Channel,`**
  `          char *Dest )`

  Reads a character from channel `Channel` to the memory pointed to by `Dest`. The receiver will be enabled automatically if CTS/RTS flow control is enabled and the receive buffer has more than 16 bytes remaining (after the read).

  PARAMETERS: `Channel` is the logical channel.

  `Dest` is the address (buffer) to read character into.

  RETURN VALUE: The actual number of bytes read from the channel.

- **`uint aascReadBlk( CHANNEL  Channel, void *Dest,`**
  `          uint Length, char Flags )`

  Reads a block of `Length` bytes from logical channel `Channel` to the memory pointed to by `Dest`. If `Flags` is non zero, either the entire `Length` or no bytes will be read. The receiver will be enabled automatically if flow control is enabled and the receive buffer has more than 16 bytes left (after the read).

  PARAMETERS: `Channel` is the logical channel.

  `Dest` is the address (buffer) to read into.

  `Length` is the number of bytes to read.

  If `Flag` is non-zero either all `Length` bytes will be read or no bytes will be read.

  RETURN VALUE: The actual number of bytes read from the channel.

- **`uint aascWriteChar( CHANNEL  Channel,`**
  `          char Src )`

  Writes a character `Src` to logical channel `Channel`. The transmitter is enabled automatically after the character is transferred.

  RETURN VALUE: The actual number of bytes written to the channel.

- **`uint aascWriteBlk( CHANNEL  Channel, void *Src,`**
  `           uint Length, char Flags )`

  Writes a block of **`Length`** bytes to logical channel **`Channel`** from the memory pointed to by **`Src`**. If **`Flags`** is non zero, either the entire **`Length`** or no bytes will be written. The transmitter is turned on automatically after the bytes are written to the buffer.

  PARAMETERS: **`Channel`** is the logical channel.

  `Dest` is the address (buffer) to write from.

  `Length` is the number of bytes to write.

  If **`Flag`** is non-zero either all **`Length`** bytes will be written or no bytes will be written.

  RETURN VALUE: The actual number of bytes written to the channel.

- **`uint aascPeek( CHANNEL Channel,`**
  `           void *pMatchee, uint size )`

  Tries to match as much data of up to size **`size`** as possible pointed to by **`pMatchee`** (not null-character terminated).

  PARAMETERS: **`Channel`** is the logical channel.

  `pMatchee` is the address of string to match.

  `size` is the number of bytes to attempt to match.

  RETURN VALUE: The number of bytes actually matched.

- **`uint aascScanTerm( CHANNEL Channel,`**
  `           char Term )`

  Scans the receive buffer of logical channel **`Channel`** for the terminating character **`Term`**. Note that this function does not read any bytes from the receive buffer. The receiver will be enabled automatically if flow control is enabled and the receive buffer has more than 16 bytes remaining.

  RETURN VALUE: The packet size terminated by **`Term`**.

- **`void aascPipe( CHANNEL Channel,`**
  `           CHANNEL Out, CHANNEL In )`

  Makes a pipe by diverting the output of **`Channel`** to the input of **`Out`**, and diverting the input of **`Channel`** from **`In`**.

  PARAMETERS: **`Channel, Out,`** and **`In`** are logical channels.

- **`long aascGetError( CHANNEL  Channel )`**

  Gets the current error condition.

  PARAMETER: **`Channel`** is the logical channel.

  RETURN VALUE: Depends on the device. For specific return values, see the description of the device driver's **`<device_name>GetErr()`** function (for example, **`sio0GetErr()`**).

- **`void aascClearError( CHANNEL  Channel )`**

  Clears the error condition.

  PARAMETER: **`Channel`** is the logical channel.

- **`uint aascReadBufLeft( CHANNEL Channel )`**

  Computes the number of bytes left to be read from the receive buffer of logical channel **`Channel`**.

  RETURN VALUE: The number of bytes left to be read.

- **`uint aascWriteBufLeft( CHANNEL  Channel )`**

  Computes the number of bytes left to be transmitted from the transmit buffer of logical channel **`Channel`**.

  RETURN VALUE: The number of bytes left to be transmitted.

- **`uint aascReadBufFree( CHANNEL  Channel )`**

  Computes the number of bytes free in the receive buffer of logical channel **`Channel`**.

  RETURN VALUE: The number of free bytes.

- **`uint aascWriteBufFree( CHANNEL  Channel )`**

  Computes the number of free bytes in the transmit buffer of logical channel **`Channel`**.

  RETURN VALUE: The number of free bytes.

- **`void aascFlush( CHANNEL Channel )`**

  Flushes the buffers associated with logical channel **`Channel`**, and loses all information that may be left in the buffers. If the channel is capable of CTS/RTS flow control, the programmer should determine whether to explicitly reenable the receive channel by calling **`aascRxSwitch`**. **`aascRxSwitch`** will disable RTS explicitly to allow the other side to transmit.

- **`void aascFlushRdBuf( CHANNEL Channel )`**

  Flushes the read buffer associated with logical channel **`Channel`**, and loses all information that may be left in the buffer. If the channel is capable of CTS/RTS flow control, the programmer should determine whether to explicitly reenable the receive channel by calling **`aascRxSwitch`**. **`aascRxSwitch`** will explicitly disable RTS to allow the other side to transmit.

- **`void aascFlushWrBuf( CHANNEL Channel )`**

  Flushes the write buffer associated with logical channel **`Channel`**. All information is erased from the buffer.

- **`void aascPrintf( CHANNEL Chan, char *fmt, … )`**

  Prints a formatted string to channel **`Chan`**.

  PARAMETERS: **`Chan`** is the channel to send to.

  `fmt` is the format of the string to be printed.

  Arguments (if any) should follow **`fmt`**.

- **`void aascVPrintf( CHANNEL Chan, char *fmt,`**
  `void *firstArg )`

  Prints a formatted string to channel **`Chan`**.

  PARAMETERS: **`Chan`** is the channel to send to.

  `fmt` is a format string.

  `firstArg` is a pointer to the first argument.

XModem Functions in AASC.LIB

The XModem protocol performs packet-based file transfers with CRC error detection.

The packet structure for XModem transfer appears below.

Bytes	Description
1	Start Of Header
1	Packet Sequence Number
1	1’s Complement of Packet Sequence Number
…	DATA (128 or 1024 bytes, binary or text)
2	CRC-CCITT (0x1021 divisor)

- **`void aascXMRdInitPhy( uint Where, uint Length,`**
  `          ulong XmemSrcAddr )`

  Initializes location and size of physical memory for **`aascReadXModem()`** PC-to-target data transfer. Specifies the location on the target and the maximum number of bytes to be transferred from the PC.

  PARAMETERS: **`Where`** is the root memory location on the target where the data being transferred are placed.

  `Length` is the maximum number of bytes to transfer.

  `XmemSrcAddr` is the final memory destination.

- **`void aascXMRdInitLog( uint Where, uint Length )`**

  Initializes location and size of logical memory for **`aascReadXModem()`** PC-to-target data transfer. Specifies the location on the target and the maximum number of bytes to be transferred from the PC.

  This default function tells the default callback read function **`aascRdCBackLocLg`** to advance the buffer pointer by the packet size after receiving each packet.

  PARAMETERS: **`Where`** is the root memory location on the target where the data being transferred are placed.

  `Length` is the maximum number of bytes to transfer.

- **`uint aascReadXModem( CHANNEL Channel,`**
  `          char *(*read_callback_loc)(),`
  `          void (*read_callback_mod)(),`
  `          char Initialize )`

  Performs XModem PC-to-target download. Call this function once with **`Initialize`** set to 1. Then set **`Initialize`** to 0, and call this function repeatedly until its return value is non zero.

  Call **`aascXMRdInitPhy()`** for physical memory transfers or **`aascXMRdInitLog()`** for logical memory transfers before using **`aascReadXModem()`**.

  PARAMETERS: **`Channel`** is the channel being read from.

  `read_callback_loc` is a pointer to a callback function that will be called by this function BEFORE each XModem packet is received. This function determines where the packet is placed in memory using the callback function **`aascRdCBackLocLg()`**.

  `read_callback_mod` is a pointer to a callback function that will be called by this function AFTER each XModem packet is received. This function performs further processing on the data. A default function **`aascRdCBackLocPh()`** that does no processing is provided.

Initialize is the initialization flag. Set **Initialize** to 1 to initialize XModem on the first call. Set **Initialize** to 0 for all subsequent calls.

RETURN VALUE:

XX_SUCCESS	**XX_TIMEOUT**
XX_COMMERR	**XX_CANCEL**
XX_SEQ	**XX_CHKSUM**
XX_NOSTART	**XX_NOBEGPAK**
XX_SYNC	

- **`uint aascRdCBackLocPh( uint PackSize,`**
  `         char PackNum )`

Dummy function called by **aascReadXModem()** after a packet is received. Can be replaced by user-defined function if modifications are required on a packet.

PARAMETERS: **PackSize** is the packet size being used by XModem (128 or 1024 bytes).

PackSize is the number of the current packet.

RETURN VALUE: Root memory logical address where packet from PC will be placed before transfer to physical memory.

- **`uint aascRdCBackLocLg( uint PackSize,`**
  `         char PackNum )`

Default callback function for addressing blocks for PC-to-target transfers. This is called by **aascReadXModem**() before receiving a packet from the PC. This function advances the pointer to the target memory by **PackSize** after each packet is sent.

PARAMETERS: **PackSize** is the packet size being used by XModem.
  0 - use 128 byte XModem packets
  1 - use 1024 byte XModem packets

PackNum is the number of the current packet.

RETURN VALUE: The logical memory address where the packet from the PC will be placed.

- **`void aascXMWrInitPhy( uint Where, uint Length,`**
  `         ulong XmemSrcAddr )`

Initializes location and size of physical memory to be transferred to the PC.

PARAMETERS: **Where** is the address on the target where the data being transferred are placed.

Length is the maximum number of bytes to receive.

XmemSrcAddr is the physical memory source of the data to transfer.

- **`void aascXMWrInitLog( uint Where, uint Length )`**

  Initializes location and size of logical memory to be transferred to the PC.

  PARAMETERS: **`Where`** is the address on the target where the data being transferred are placed.

  `Length` is the maximum number of bytes to receive.

- **`int aascWriteXModem( CHANNEL Channel,`**
  `char Pak1K, char Initialize,`
  `uint(*write_callback)() )`

  Performs XModem target-to-PC upload. Call this function once with **`Initialize`** set to 1. Then set **`Initialize`** to 0, and call this function repeatedly until it's return value is non-zero.

  Call **`aascXMWrInitPhy()`** for physical memory transfers or **`aascXMWrInitLog()`** for logical memory transfers before the first call to **`aascReadXModem()`**.

  PARAMETERS: **`Channel`** is the logical channel being written to.

  `Pak1K` is the XModem packet size.
  0 - use 128 byte XModem packets
  1 - use 1024 byte XModem packets

  `Initialize` is the initialization flag. Set **`Initialize`** to one to initialize XModem on the first call. Set **`Initialize`** to zero for all subsequent calls.

  `write_callback` is a pointer to a callback function that will be called by this function BEFORE each XModem packet is sent so that further processing can be performed on the data. The default functions **`aascWrCallBackLg()`** and **`aascWrCallBackPh()`** are provided for logical and physical memory transfers. See on-line help on these functions for further details.

  RETURN VALUE:

  ```
  XX_SUCCESS
  XX_TIMEOUT
  XX_COMMERR
  XX_CANCEL
  XX_NOSTART
  XX_SYNC
  ```

- **`uint aascWrCallBackPh( uint PackSize,`**
  `           char PackNum )`

  Default callback function for addressing data for target-to-PC transfers. **`aascWrCallBackPh`** is called by **`aascWriteXModem()`** before sending a packet to the PC. **`aascWrCallBackPh`** advances the pointer to the target's memory by **`PackSize`** after each packet is sent. **`aascWrCallBackPh`** determines the address based on **`PackSize`** and **`Packnum`**.

  PARAMETERS: **`PackSize`** is the packet size being used by XModem.
  0 - use 128 byte XModem packets
  1 - use 1024 byte XModem packets

  `PackNum` is the number of the current packet.

  RETURN VALUE: The address of the next location to transfer data from; 0 if the requested packet number exceeds the file size.

- **`uint aascWrCallBackLg( uint PackSize,`**
  `           char PackNum )`

  Default callback function for addressing data for target-to-PC transfers. **`aascWrCallBackLg`** is called by **`aascWriteXModem()`** before sending a packet to the PC. **`aascWrCallBackLg`** advances the pointer to the target's memory by **`PackSize`** after each packet is sent. **`aascWrCallBackLg`** determines the address based on **`PackSize`** and **`Packnum`**.

  PARAMETERS: **`PackSize`** is the packet size being used by XModem.
  0 - use 128 byte XModem packets
  1 - use 1024 byte XModem packets

  `PackNum` is the number of the current packet.

  RETURN VALUE: The address of the next location to transfer data from; 0 if the requested packet number exceeds the file size.

5 OTHER LIBRARIES

The libraries described in Chapter 5 are specific to one or more types of controllers.

5KEY.LIB

These LCD and keypad functions support the PK2100 and PK2200 series controllers. This is the ***old five-key system***. It uses the real-time kernel (RTK). The standard LCD is 2 × 20. To run the five-key system with a 2 × 16 LCD, write **#define LCD16x2** at the start of the program.

- **`void _5keysettime( char *time )`**

  Sets real-time clock time, based on string **`*time`**. The string format is “hh:mm:ss”.

- **`void _5keysetdate( char *date )`**

  Sets real-time clock date, based on string pointed to by ***date**. The string format is “mm-dd-yy”.

- **`void _5keygettime( char *time )`**

  Gets real-time clock time and stores it in **`*time`**. The string format is “hh:mm:ss”.

- **`void _5keygetdate( char *date )`**

  Gets real-time clock date and stores it in **`*date`**. The string format is “mm-dd-yy”.

- **`void lcd_server( char mode, long position, char *lcd_msg )`**

  Clears number of lines, specified by **`mode`**, and displays message **`*lcd_msg`** at **`position`**. See **`CPLC.LIB`** for description of position fields.

-
  ```
  int _5key_float(
      char *label,  float *value,
      float max,    float  min,
      char *help[], byte   size,
      byte  modify, byte   delay )
  ```

  This is the five-key system handler for a float parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`value`	pointer to a **`float`** variable
`max, min`	the data limits
`help[]`	an array of help strings
`size`	size of the help array (the number of help strings); use **`sizeof(help)`**
`modify`	if 1, **`value`** is updated; if 0, **`value`** is only monitored

delay	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_integer(`**
```
    char *label,  int *value,
    int   max,    int  min,
    char *help[], byte size,
    byte  modify, byte delay )
```

This is the five-key system handler for an integer parameter. It modifies or monitors the following parameters.

label	the item label (string)
value	pointer to an integer variable
min, max	the data limits
help[]	an array of help strings
size	size of the help array (the number of help strings); use **sizeof(help)**
modify	if 1, **value** is updated; if 0, **value** is only monitored
delay	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_boolean(`**
```
    char *label,  byte *value,
    char *help[], byte  size,
    byte  modify, byte  delay )
```

This is the five-key system handler for a Boolean parameter. It modifies or monitors the following parameters.

label	the item label (string)
value	pointer to a “Boolean” variable
help[]	an array of help strings
size	size of the help array (the number of help strings); use **sizeof(help)**
modify	if 1, **value** is updated; if 0, **value** is only monitored

`delay`	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_time(`**
```
    char *label,   char *string,
    char *help[],  byte  size,
    byte  set_clock,
    byte  modify,  byte  delay )
```

This is the five-key system handler for a time parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`string`	the time string
`help[]`	an array of help strings
`size`	size of the help array (the number of help strings); use `sizeof(help)`
`set_clock`	if non-zero, set the real-time clock
`modify`	if 1, `value` is updated; if 0, `value` is only monitored
`delay`	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_date(`**
```
    char *label,   char *string,
    char *help[],  byte  size,
    byte  set_clock,
    byte  modify,  byte  delay )
```

This is the five-key system handler for a date parameter. It modifies or monitors the following parameters.

`label`	the item label (string)
`string`	the date string
`help[]`	an array of help strings
`size`	size of the help array (the number of help strings); use `sizeof(help)`
`set_clock`	if non-zero, set the real-time clock

modify	if 1, **value** is updated; if 0, **value** is only monitored
delay	number of 25-millisecond RTK ticks after which the software will release the current five-key task, freeing other lower priority tasks.

The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

-
  ```
  void _5key_setmenu(
      char *menu,      char *item,
      byte  mode,      void *ptr,
      float max,       float min,
      char *help[],    byte  size,
      byte  modify,    byte  delay,
      byte  display )
  ```

  Adds a five-key item to the five-key linked list. Items with the same menu label are grouped together. The following parameters are used.

menu	the menu label (string)
item	the item label (string)
mode	type of data being created
ptr	pointer to the data
max, min	the data limits
help[]	array of help strings
size	size of the help array (the number of help strings); use **sizeof(help)**
modify	determines the handling of the data. To modify or to monitor.
delay	number of RTK ticks (25 ms) after which the software will release the current five-key task, allowing other lower priority tasks to execute
display	1 if item is to be added to the list of periodically displayed items; 0 if item is not to be added to the list.

-
  ```
  int _5key_init_item(
      _5KEYITEM *thisitem, char *d_menu,
      char *d_item, char data_mode,
      void *data_ptr, float max_data,
      float min_data, char *my_help[],
      char help_line, char data_modify,
      char delay )
  ```

  Is called by **_5key_setmenu** to create a five-key item. The following parameters are used.

thisitem points to a five-key item structure for the five-key link list

d_menu points to a menu label

d_item points to an item label

data_mode is 0 for floats; 1 for **ints**; 2 for boolean (**char**s); 3 for time strings; 4 for date strings.

The following macros can also be used.

_**5key_Fdata**, **_5key_Idata**, **_5key_Bdata**, **_5key_Tdata** and **_5key_Ddata.data_ptr** point to the data

max_data is the upper limit and **min_data** is the lower limit for the data

my-help[] is a list of help strings

help-line is twice the actual number of help strings

data_modify is 1 if data are to be modified through the five-key system; else 0, if data are just monitored

delay is the five-key task suspend period

idisp is 1 if data are to be displayed periodically when there are no keypad and lcd activities; else 0.

The function returns None.

- **`int _5key_server( _5KEYITEM *t_item )`**

Services a five-key item for display to the LCD and actions. The function returns any of the five-key menu keys pressed.

- **`void _5key_menu()`**

Services the linked list created with **`_5key_setmenu()`**. This function must be called inside an RTK task.

- **void _5key_setalarm (**

```
void _5key_setalarm (
    int(*func1)(), int(*func2)(),
    int(*func3)(), int(*func4)() )
```

Sets up the service functions for the software alarms.

func1(, the service function for _ALARM1
func2(), the service function for _ALARM2
func3(), the service function for _ALARM3, and
func4(), the service function for _ALARM4.

All the functions default to **NO_FUNCTION**. Service functions can be changed or turned off at run-time as long as there is no conflict with the execution of a service function.

- **`void _5key_setfunc (`**
  `int(*func1)(), int(*func2)(),`
  `int(*func3)(), int(*func4)() )`

  Sets up the service functions for the function keys.

  `func1()`, the service function for F1
  `func2()`, the service function for F2
  `func3()`, the service function for F3, and
  `func4()`, the service function for F4.

  All the functions default to **`NO_FUNCTION`**. Service functions can be changed or turned off at run-time as long as there is no conflict with the execution of a service function.

- **`void _5key_setmsg( byte message_no,`**
  `char *the_message )`

  Sets one of ten message strings for periodic display.

`message_no`	the message number, 0–9
`the_message`	the message string.

  All the messages default to NULL.

5KEYEXTD.LIB

These keypad functions support the PK2100 and PK2200 series controllers. They use the real-time kernel (RTK).

- **`int _5key_12out()`**

  This is the five-key server for the ten “virtual” digital outputs and two “virtual” relay outputs. The digital output and the relay output states can be modified through the five-key system. If an output state changes, this function will refresh the display to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_dacout()`**

  This is the five-key server for the “virtual” DAC channel. If the output value changes, this function will refresh the screen to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_uinput()`**

  This is the five-key server for the six “virtual” universal inputs. If an input state changes, this function will refresh the display to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_diginput()`**

  This is the five-key server for the seven “virtual” digital inputs. If an input state changes, this function will refresh the screen to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_bank1dig()`**

  This is the five-key server for the “virtual” digital inputs 1–8. If an input state changes, this function will refresh the screen to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_bank2dig()`**

  This is the five-key server for the “virtual” digital inputs 9–16. If an input state changes, this function will refresh the screen to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

- **`int _5key_14out()`**

  This is the five-key server for the 14 “virtual” digital outputs. The digital output states can be modified through the five-key system. If an output state changes, this function will refresh the display to reflect the change.

  The function returns an integer representing one of the following keys: MENU, ITEM, ADD or DELETE. It returns –1 when no key has been pressed.

CPLC.LIB

These functions support the PK2100 and PK2200 series controllers.

- **`void uplc_init()`**

  Initializes drivers and variables for the following.

  > interrupt routine for background timer 1
  > LCD, when selected
  > keypad, if selected (keypad is scanned at 25 milliseconds)
  > virtual drivers, virtual timers and virtual watchdogs, when selected.

  The timer 1 interrupt routine also services the watchdog timer.

- **`void lc_kxinit()`**

  Initializes keypad driver and associated variables as well as virtual watchdog variables.

- **`void up_beepvol( int vol )`**

  Sets beeper volume: **`vol`** = 1 for low volume; 2 for high volume.

- **`void lc_loadtab( int *tab, int tab_size )`**

  Loads **`tab`** tables to match LCD screen.

- **`void lc_settab( char flag )`**

  Sets the tab variable **`lc_usetab`**.

- **`int lc_kxget( char mode )`**

  Fetches key value from FIFO keypad buffer. If **`mode`** = 0, value is removed from buffer; else value remains in buffer.

  The function returns the key value, or –1 if no key was pressed.

- **`void lc_setbeep( int delay )`**

  Sets beeper duration for **`delay`** counts of 1280-Hz cycles.

- **`void up_beep( uint k )`**

  Sets beeper on for **`k`** milliseconds.

- **`uint up_lastkey()`**

  Returns time since last key was pressed, in units of 1/40 second. The function returns elapsed time.

- **`void lc_init_keypad()`**

  Initializes **`timer1`**, keypad driver and variables, and the real-time kernel.

- **`void GLOBAL_INIT()`**

  Refere to **`VDRIVER.LIB`** for a description of this function.

- **`int up_synctimer()`**

  Synchronizes the virtual **`SEC_TIMER`** with the real-time clock (RTC). The function returns 0 if RTC is read properly, and –1 otherwise.

DRIVERS.LIB

These are miscellaneous hardware drivers.

- **`int plcport( int bit )`**

  Checks the specified bit of the PLCBus port. The function returns 1 if the specified bit is set, or 0 if not.

- **`void set16adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 16-bit physical address (for 4-bit bus). The high-order nibble contains the expansion register value, while the remaining nibbles form a 12-bit address (the first and third nibbles must be swapped).

- **`void set12adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 12-bit physical address (for 4-bit bus) with the first and third nibbles swapped (most significant nibble are in the low four bits).

- **`void set4adr( int address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** contains the last 4-bits of the physical address (for 4-bit bus) in bits 8–11. A 12-bit address may be passed to this function, but only the last 4 bits will be set. This function should only be called if the first 8 bits of the address are the same as the address in the previous call to **`set12adr`**.

- **`char read4data( int address )`**

  Sets the last 4 bits of the current PLCBus address using **`address`** (bits 8–11). Then reads 4 bits of data off of the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 4 bits (upper bits are undefined).

- **`char read12data( int address )`**

  Sets the current PLCBus address using the 12-bit **`address.`** Then reads 4 bits of data off of the bus with a **`BUSRD0`** cycle.

  The function returns PLCBus data in the lower 4 bits (upper bits are undefined).

- **`void write4data( int address, char data )`**

  Sets the last 4 bits of the current PLCBus address using **`address`** (bits 8–11). Then writes the low 4 bits of **`data`** to the bus.

- **`void write12data( int address, char data )`**

  Sets the current PLCBus address using the 12-bit **`address.`** Then writes the low 4 bits of **`data`** to the bus.

- **`void hv_wr( char v )`**

  Writes 8 bits to the high-voltage driver. Each bit affects one high-voltage output. A 1 enables the corresponding output; 0 disables the output.

- **`void hv_enb()`**

  Enables high-voltage driver.

- **`void hv_dis()`**

  Disables high-voltage driver.

- **`void lcd_init( char mode )`**

  Initializes the LCD; **`mode`** should normally be set to 0x18.

- **`void lputc( char cc )`**

  Sends a character to the LCD and updates the cursor (without wrap-around); **`cc`** is the character to send: if the high bit is set, it will be treated as a control character. Possible control characters are as follows.

\n	Newline (position cursor to line 1, column 0)
\xFF	Clear screen
\xF0	Clear line 0
\xF1	Clear line 1
\xF2	Cursor OFF (cursor invisible, blink off)
\xF3	Cursor ON (solid cursor block)
\xF4	Cursor BLINK (blinks continuously)
\xF5	Shift display left
\xF6	Shift display right
\x80–\xA7	Position cursor at line 0
\xC0–\xE7	Position cursor at line 1

- **`void lcd_clr_line( char code )`**

  Clears a line on the LCD; **`code`** should be 0x80 to clear line 0 and 0xC0 to clear line 1.

- **`void lcd_wait()`**

  Waits until the LCD is ready to accept data.

- **`int lprintf( char *fmt, ... )`**

  Operates the same as **`printf`**, but outputs to LCD.

- **`char *lputs( char *p )`**

  Sends the null-terminated string **`*p`** to the LCD and updates the cursor (without wraparound). All characters (except null) are sent directly to the LCD; control characters are not recognized. The function returns a pointer to the string.

- **`void* intoff( void* ptr )`**

  Saves the current interrupt state in **`*ptr`** and then disables interrupts. The function returns the pointer **`ptr`**.

- **`void* inton( void* ptr )`**

  Enables interrupts if they were previously on, according to **`*ptr`**. **`ptr`** must have been set previously by a call to **`intoff`**. The function returns the pointer **`ptr`**.

- **`void doint()`**

  Enables interrupts for a short time and then disables them (if they were previously off). This allows interrupts to be processed in code where they are otherwise disabled.

- **`int tm_rd( struct tm *t )`**

  Reads the current system time into the structure **`t`**. This routine works with either the Toshiba or Epsom clocks.

  The following structure is used to hold the time and date:

  ```
  struct tm {
       char tm_sec;       // 0-59
       char tm_min;       // 0-59
       char tm_hour;      // 0-23
       char tm_mday;      // 1-31
       char tm_mon;       // 1-12
       char tm_year;      // 00-150 (1900-2050)
       char tm_wday;      // 0-6 where 0 means Sunday
    };
  ```

  The function returns 0 if successful, and –1 if the clock is failing or is not installed.

- **`int tm_wr( struct tm *t )`**

  Sets the system time according to the structure **`t`**. This routine works with either the Toshiba or Epsom clocks.

  The following structure is used to hold the time and date:

```
  struct tm {
       char tm_sec;       // 0-59
       char tm_min;       // 0-59
       char tm_hour;      // 0-23
       char tm_mday;      // 1-31
       char tm_mon;       // 1-12
       char tm_year;      // 00-150 (1900-2050)
       char tm_wday;      // 0-6 where 0 means Sunday
    };
```

  The function returns 0 if successful, and –1 if the clock is failing or is not installed.

- **`void mktm( struct tm *timeptr, long time )`**

  Fills the structure pointed to by **`timeptr`** according to time, specified in seconds since January 1, 1980.

- **`long mktime( struct tm *timeptr )`**

  Converts the contents of **`timeptr`** into a long integer. The function returns time in seconds since January 1, 1980.

- **`long clock()`**

  Reads the system clock and converts time to a long integer. The function returns system time in seconds since January 1, 1980.

- **`long phy_adr( char *adr )`**

  Converts a logical (16-bit) address to a physical (20-bit) address. **`adr`** points to the address. The function returns 20-bit address as a long integer.

- **`void dmacopy( long dest, long src, uint count )`**

  Uses DMA to copy **`count`** bytes from one physical address (**`src`**) to another (**`dest`**).

- **`void outportn( int port, char *buf, char count )`**

  Writes **`count`** bytes to the specified output port. **`buf`** points to the sequence of bytes to write.

- **`void init_timer0( uint count )`**

  Initializes timer 0. **`count`** is the value placed in the reload register. Some common count values and the frequencies they generate are provided below for a 9.216-MHz clock.

9126	50 Hz	7680	60 Hz	7200	64 Hz
4608	100 Hz	2304	200 Hz	1152	400 Hz
900	512 Hz	600	768 Hz	500	928 Hz
450	1024 Hz				

- **`void timer0_isr()`**

  `timer 0` interrupt service routine, runs the real-time kernel.

- **`void setbeep( int delay )`**

  Sets up a timed beep. **`delay`** specifies the length of the beep in number of **`timer1`** ticks. **`timer1`** interrupt performs the beep in the background, so this function returns immediately.

- **`void init_timer1( uint count )`**

  Initializes **`timer1`**. **`count`** is the value placed in the reload register. Some common count values and the frequencies they generate are provided below for a 9.216-MHz clock.

9126	50 Hz	7680	60 Hz	7200	64 Hz
4608	100 Hz	2304	200 Hz	1152	400 Hz
900	512 Hz	600	768 Hz	500	928 Hz
450	1024 Hz				

- **`void tdelay( int msec )`**

  Waits for **`msec`** milliseconds, assuming that **`timer1`** is running at 750 Hz. The actual delay is related to the frequency of **`timer1`** by the formula delay = 3 × (**`msec`**/4)/**`freq1`**.

- **`void int_timer1()`**

  `timer1` interrupt service routine. Drives the beeper and keypad. Also runs the real-time kernel if **`RUNKERNEL`** is defined.

- **`void save_shadow()`**

  Saves PLCBus shadow registers on the stack.

- **`void restore_shadow()`**

  Restores PLCBus shadow registers from the stack and resets the current bus address.

- **`void write24data( long address, char data )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then writes 8 bits of **`data`** to the bus.

- **`void write8data( long address, char data )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then writes 8 bits of **`data`** to the bus.

- **`int read24data0( long address )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then reads 8 bits of data off of the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`int read8data0( long address )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then reads 8 bits of data from the bus with a **`BUSRD0`** cycle. The function returns PLCBus data in the lower 8 bits, with the upper bits 0.

- **`int read24data1( long address )`**

  Sets the current PLCBus address using the 24-bit **`address`**, then reads 8 bits of data from the bus with a **`BUSRD1`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`int read8data1( long address )`**

  Sets the last 8 bits of the current PLCBus address using **`address`** (bits 16–23), then reads 8 bits of data from the bus with a **`BUSRD1`** cycle. The function returns PLCBus data in the lower 8 bits (upper bits are 0).

- **`void set24adr( long address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** is a 24-bit physical address (for the 8-bit bus), with the first and third bytes swapped (low byte most significant).

- **`void set8adr( long address )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`address`** contains the last 8 bits of the physical address (for the 8-bit bus) in bits 16–23. A 24-bit address may be passed to this function, but only the last 8 bits will be set. This function should only be called if the first 16 bits of the address are the same as the address in the previous call to **`set24adr`**.

- **`void plcbus_isr()`**

  This function is used to service all PLCBus /AT line interrupts. The /AT line is connected to INT1 of the Z180. Each interrupt service routine (ISR) is responsible for assuring its device releases the /AT signal once the ISR has been performed.

- **`void relocate_int1()`**

  Reprograms the **`INT1`** vector.

- **`int DelayTicks( CoData *pfb, uint ticks )`**

  Provides tick time mechanism for costatements. Ticks occur 1280 times per second. (The period is 781.25 microseconds.) The function returns 1 if the specified tick delay has lapsed. Otherwise, it returns 0.

- **`int DelayMs( CoData *pfb, long delayms )`**

  Provides millisecond time mechanism for costatements. The function returns 1 if the specified millisecond delay has lapsed. Otherwise, it returns 0.

- **`int DelaySec( FuncBlk *pfb, long delaysec )`**

  Provides second time mechanism for costatements. The function returns 1 if the specified second delay has lapsed. Otherwise, it returns 0.

- **`int eei_rd( int address )`**

  Reads two consecutive byte areas of the EEPROM for integer data. The low byte is from **`address`** and the high byte is from **`address+1`**.

  The function returns the integer at **`address`** from EEPROM.

- **`int eei_wr( int address, uint value )`**

  Writes an integer value to the EEPROM at **`address`**. The lower byte is at **`address`** and the high byte is at **`address+1`**.

  The function returns 0 if the write was successful.

- **`void DMA0( uint cnt )`**

  Loads **`cnt`** to DMA0 counter to count high-speed pulses in hardware. Maximum count is 64,000. **`_DMAFLAG0`** is set to 0. If the DMA has counted out, the interrupt service routine for DMA0 will generate an interrupt in which **`_DMAFLAG0`** is set to 1. Events are edge sensed. C1A and C1B must both be low for /DREQ0 to generate an interrupt.

- **`void DMA1( uint cnt )`**

  Loads **`cnt`** to the DMA1 counter to count high-speed pulses in hardware. Maximum count is 64,000. **`_DMAFLAG1`** is set to 0. If the DMA has counted out, the interrupt service routine for DMA1 will generate an interrupt in which **`_DMAFLAG1`** is set to 1. Events are edge sensed. C2A and C2B must both be low for /DREQ1 to generate an interrupt. C2B uses one of the RS-485 receivers for differential input. For example, tie C2B– to 5 volts; when the signal at C2B+ is lower than 5 volts, a negative edge is generated for the DMA counter.

- **`uint DMASnapShot( char channel, uint *count )`**

  Takes a "snap shot" of a DMA **`channel`** (0 or 1) for the number of pulses counted. The function returns 0 if the pulse train is too fast to have a snapshot taken; or 1 if a snapshot is obtained and valid data is in **`*count`**.

- **`void powerdown()`**

  Turns the power off. Power can only be turned back on by external means. This only works for boards with a switching power supply (except for the PK2200).

- **`void powerup()`**

  Reverses the effect of powerdown so power stays on after external power is disabled. See **`powerdown`**.

- **`void nmiint()`**

  Default power-fail interrupt handler. The function does nothing and ***never returns***.

- **`void setperiodic( int period )`**

  Sets a timer to periodically power up the BL1100. After this call, the board may be put to sleep and will automatically awaken at the specified interval. Execution will begin in the main function when power is restored. **`period`** may be 4 (to wake once per second), 8 (to wake once per minute), or 12 (to wake once per hour). Works only for boards that have a switching power supply, except the PK2200.

- **`void sleep()`**

  Puts the controller to sleep. Works for all boards that use a switching power supply, except the PK2200.

  The function does not return.

- **`void init_timer()`**

  Initializes the system clock.

DMA.LIB

These functions support DMA use on all Z-World controllers.

- **`void DMA0Count( uint count )`**

  Loads **`count`** to DMA0 counter to count high-speed pulses in hardware. The maximum count is 64,000. The function sets the flag **`_DMAFLAG0`** to 0. DMA0 causes an interrupt when **`count`** negative edges have been detected. The interrupt service routine for DMA0 will set **`_DMAFLAG0`** to 1. A user program can monitor **`_DMAFLAG0`** to determine whether **`count`** has finished.

- **`void DMA1Count( uint count )`**

  Loads **`count`** to DMA1 counter to count high-speed pulses in hardware. The maximum count is 64,000. The function sets the flag **`_DMAFLAG1`** to 0. DMA1 will cause an interrupt when **`count`** negative edges have been detected. The interrupt service routine for DMA1 will set **`_DMAFLAG1`** to 1. A user program can monitor **`_DMAFLAG1`** to determine whether the count has finished.

- **`uint DMASnapShot( byte channel, uint *count )`**

  Reads the number of pulses that a DMA channel (**`channel`** = 0 or 1) has counted. A DMA counter is initialized with either **`DMA0Count`** or **`DMA1Count`**. The function returns 0 if a DMA channel is counting too fast to allow a stable reading of the **`count`** value. If the function reads a stable `count` value, it returns 1 and sets the parameter **`*count`**. Note that a DMA interrupt will still occur when the DMA channel finishes counting, even if the **`count`** cannot be read.

- **`void DMA0_Off()`**
  `void DMA1_Off()`

  Turns the DMA channel off.

- **`uint DMA0_SerialInit( byte channel, byte mode, byte baud )`**

  Initializes serial port **`channel`** (must be 0 or 1) of the Z180 for DMA0 to serial transfers.

  The term **`mode`** is defined as follows.

bit0 = 0 for 1 stop bit	1 for 2 stop bits
bit1 = 0 for no parity	1 for parity
bit2 = 0 for 7 data bits	1 for 8 data bits
bit3 = 0 for even parity	1 for odd parity.

  The term **`baud`** is the baud rate in multiples of 1200 baud (e.g., 8 for 9600 baud).

- **`uint DMA0_Rx( byte port, ulong address, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to receive **`count`** bytes from a serial port (**`port`** = 0 or 1) to absolute memory locations starting at **`address`**. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays. DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 does not generate an interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

The function returns 1 if successful, 0 if DMA0 is busy, –1 if the serial port is busy, and –2 if `channel` is not 0 or 1.

- **`uint DMA0_Tx( byte port, ulong address, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to transmit `count` bytes to a serial port (`port` = 0 or 1) from absolute memory locations starting at `address`. The logical memory address for ordinary arrays may be converted to a physical address with `phy_adr(array)`. Simply pass the array name directly for `xdata` arrays. DMA0 will generate an interrupt at the end of the transfer if `interrupts` is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if `interrupts` is 0. The term `increments` must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, 0 if DMA0 is busy, –1 if the serial port is busy, and –2 if `channel` is not 0 or 1.

- **`uint DMA0_MM( ulong dst, ulong src, uint count, int mode, int interrupts )`**

  Initiates a transfer using DMA0 to copy `count` bytes from absolute memory locations starting at `src` to absolute memory locations starting at `dst`. The logical memory address for ordinary arrays may be converted to a physical address with `phy_adr(array)`. Simply pass the array name directly for `xdata` arrays. DMA0 will generate an interrupt at the end of the transfer if `interrupts` is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if `interrupts` is 0. The term `mode` must be 0 for cycle-stealing transfers, and 1 for burst transfers.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`uint DMA0_MIO( uint ioaddr, ulong memaddr, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to write `count` bytes from absolute memory locations starting at `memaddr` to the I/O port designated by `ioaddr`. The external device must generate negative-going /DREQ0 pulses for each byte transferred. The logical memory address for ordinary arrrays may be converted to a physical address with `phy_adr(array)`. Simply pass the array name directly for `xdata` arrays. DMA0 will generate an interrupt at the end of the transfer if `interrupts` is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if `interrupts` is 0. The term `increments` must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`uint DMA0_IOM( ulong memaddr, uint ioaddr, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA0 to read **`count`** bytes from the I/O port designated by **`ioaddr`** to the absolute memory locations starting at **`memaddr`**. The external device must generate negative-going /DREQ0 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays. DMA0 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA0 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA0 is busy.

- **`uint DMA1_MIO( uint ioaddr, ulong memaddr, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA1 to write **`count`** bytes from absolute memory locations starting at **`memaddr`** to the I/O port designated by **`ioaddr`**. The external device must generate negative-going /DREQ1 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays. DMA1 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA1 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA1 is busy.

- **`uint DMA1_IOM( ulong memaddr, uint ioaddr, uint count, int interrupts, int increments )`**

  Initiates a transfer using DMA1 to read **`count`** bytes from the I/O port designated by **`ioaddr`** to the absolute memory locations starting at **`memaddr`**. The external device must generate negative-going /DREQ1 pulses for each byte transferred. The logical memory address for ordinary arrays may be converted to a physical address with **`phy_adr(array)`**. Simply pass the array name directly for **`xdata`** arrays. DMA1 will generate an interrupt at the end of the transfer if **`interrupts`** is 1. The user program must provide the interrupt service routine. DMA1 generates no interrupt if **`interrupts`** is 0. The term **`increments`** must be 0 to increment the memory address, and 1 to decrement the memory address.

  The function returns 1 if successful, and 0 if DMA1 is busy.

FK.LIB

These are LCD and keypad support functions for use without the real-time kernel (RTK).

- **`int fk_helpmsg( char **hptr )`**

  Displays a series of help messages when the HELP key is pressed. The current display is saved and each message string is displayed for 1.8 seconds, then the previous display is restored. The input should be an array of strings declared like this.

  ```
  char *hptr[]={"Str 1","Str 2",...,"StrN",""};
  ```

  The last string must be ***null***. The function returns non-zero if help is off, and zero if help is on.

- **`void fk_monitorkeypad()`**

  Monitors the keypad for keys pressed. This function should be called from an SRTK or RTK high-priority task. It sets global variable **`fk_tkey`** to values from 1 to 12 depending on the key pressed. The value is 0 if no key was pressed. The function also monitors for the 2-key reset combination. If a reset combination is detected, the unction will not return but will force a watchdog timeout. There is no buffer. Key presses should be processed within 100 milliseconds or they will be lost.

- **`int fk_item_alpha( char *s1, char *var, int wdsize )`**

  Modifies a string using the five-key system. The term **`*s1`** is a string containing a prompt. The term **`*var`** is the string to be displayed and/or modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_int( char *string, int *num, int lower, int upper )`**

  Displays/modifies an integer number using the five-key system. The term **`*string`** is a **`printf`** format having the form **`%`*`n`*`u`** where *n* is 1 digit, for example, **`%5d`**. The term **`*num`** is the integer to be displayed and/or modified. The arguments **`upper`** and **`lower`** are the upper and lower limits for the number. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_uint( char *string, uint *num, uint lower, uint upper )`**

  This function is the same as **`fk_item_int`**, but applies to unsigned integers. (Remember that **`uint`** is a convention in this manual only and is not a C keyword.)

- **`int fk_item_float( char*s1, float *num, float lower, float upper )`**

  Displays/modifies a floating-point number using the five-key system. The term **`*s1`** is a **`printf`** format for displaying the number. The format code should be in the form of **`%n.mf`**. The displayed line appears as follows.

  ```
  vvvvvv wwww.yyyy
  ```

  where **`vvvvv`** is a prompt string, **`wwww`** is *n* chars long, and **`yyyy`** is *m* chars long. The value *n* must be at least 1. The sum *n* + *m* cannot exceed 9. The default is *n* = 5 and *m* = 2. The term **`*num`** is the floating-point number to be displayed and/or modified. The arguments **`upper`** and **`lower`** are the upper and lower limits for the number. This function will work for numbers in the ranges [1E6,–1E–4], [1E–4,1E6] with the appropriate format specification.

  The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_enum( char *prompt, int *choice, char *s1,...*sn, "")`**

  Allows the user to choose from a list of null-terminated strings (maximum 20). The string **`*prompt`** must contain a string field code (**`%s`** or **`%ns`**) used to print the strings. The last of the strings (after **`*s1`**, ... **`*sn`**) must be ***null***. The term **`*choice`** returns the choice made by the user, from 0 to (n - 1). The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_setdate( struct tm *time )`**

  A five-key function to modify the day, month and year fields of a **`tm`** structure. The term **`*time`** is the structure to be modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

- **`int fk_item_settime( struct tm *time )`**

  A five-key function to modify the hour, minute and second fields of a **`tm`** structure. The term **`*time`** is the structure to be modified. The function returns 0 if not done, and 1 if done, and returns 1 or 0 in global variable **`fk_newmenu`**.

XP88XX.LIB

These are stepper motor support functions.

- **`uint sm_bdaddr( int jumpers )`**

  Returns the PLCBus address for a stepper motor card for the given jumper settings, as defined at H4. The function returns PLCBus address for card.

- **`int sm_poll( uint bdaddr )`**

  Given a PLCBus address for a stepper motor card, the function returns 0 if the board is found, and 1 if not.

- **`void sm_hitwd( int index )`**

  Hits the MAX705 watchdog by executing a counter-read cycle. **`index`**: is the expansion board index.

- **`int sm_find_boards()`**

  Polls all 16 possible motor board addresses and loads the addresses of those found into the array **`sm_addr`**. The addresses found will be stored in the array in order from lowest jumper setting found to highest (0–15). Corresponding arrays of status bytes and service flags are initialized.

  Returns the number of boards found. The element following the last address found is set to **`0xFFFF`**. The control register is initialized to a value of **`0xA7`** on all boards found. The function returns an integer representing the number of stepper motor boards that respond to the poll.

- **`uint smq_read16( int index )`**

  Returns the entire 16-bit number of the quadrature counter. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. The function returns the 16-bit value of the quadrature counter.

- **`char smq_read8( int index )`**

  Returns the low byte of the quadrature counter. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. The function returns the lower byte of the counter.

- **`void sm_board_reset( int index )`**

  Performs a hardware reset on the controller and encoder. Disables the driver and sets it to dual-phase mode. Sets the register select lines to 00. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void smq_hardreset( int index )`**

  Sends a hardware reset command to the quadrature counter. Resets the counter to zero. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void smc_hardreset( int index )`**

  Sends a hardware reset command to the PCL-AK. This stops the output and clears the internal registers. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void smc_softreset( int idx )`**

  Sends a software reset command to the PCL-AK. This stops the output without clearing the internal registers. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void smc_cmd( int index, int cmd )`**

  Writes to the command register in the PCL-AK controller. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void smc_setspeed( int index, int s1, int s2 )`**

  Sets the high and low speed registers to the given numbers. The multiplier register is set to 732 to make the speed values in pulses per second. The pulse output is set to on, no ramp-down IRQ, and normal polarity. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`s1`** is the fast rate in pulses per second, and **`s2`** is the slow rate.

- **`void smc_manual_move( int index,int dir, int speed )`**

  Starts a manual move operation. The motor will move until a decelerating stop command, a software reset (**`smc_softreset`**) is issued, or an EL or ORG pulse is detected (if enabled). **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`dir`** is the move direction: 1 = forward, 0 = backward; **`speed`**: 1 = fast mode (R2 value), 0 = slow mode (R1 value).

- **`void smc_seek_origin( int index,int dir, int speed )`**

  Moves motor until an origin pulse is detected. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`dir`** is the move direction: 1 = forward, 0=backward; **`speed`**: 1 = fast mode (R2 value), 0 = slow mode (R1 value).

- **`void smc_setmove( int index,long R0,int R1, int R2,int R4,int R6,int R7 )`**

  Sets up the registers of the PCL-AK for a move operation. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`R0`**: number of pulses to move; **`R1`**: low-speed move rate; **`R2`**: high-speed move rate; **`R4`**: acceleration rate; **`R6`**: ramp-down point; **`R7`**: multiplier register (set to 732 for **`R1`** and **`R2`** in pulses per second).

- **`uint smcq_moveto( int index, uint dest, int dir, uint acc )`**

  Manually moves the motor until the encoder reaches a given value. The move is done at the rate as specified in **`R1`** (slow rate) of the controller. For example,

  ```
    smcq-moveto(myaddr, 5000, 1, 25);
  ```

  moves forward until the encoder is read in the range 4075–5025.

The move speed, encoder resolution, and motor degrees/phase will effect how tightly the accuracy may be confidently applied. It is possible to miss a stop point if too tight an accuracy is applied. The encoder should be read after the operation (allowing time for the motor to come to a stop) to ensure its location is accurate.

`index` is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`dest`** is the encoder value to stop at; **`dir`** is the move direction: 1 = forward, 0 = backward; **`acc`** is the accuracy of the stop value.

The function returns the last encoder value read (when the decision to stop was made). Inertia and step locations will make the final resting place value differ from this number.

- **`char smc_stat0( int index )`**

  Reads the status register at address 0 (A1 = A0 = 0) on the PCL-AK controller. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. The function returns the value from the STAT0 register on the PCL-AK.

- **`char smc_stat3( int index )`**

  Reads the status register at address 1 (A1 = A0 = 1) on the PCL-AK controller. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. The function returns the value from the STAT3 register on the PCL-AK.

- **`void sm_ctlreg( int index, int dat )`**

  Writes a value **`dat`** to the write-only control register on the stepper motor expansion card. Updates the shadow variable for register. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void sm_drvoe( int index,int onoff )`**

  Turns the motor driver output on or off. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`onoff`**: 0: off, 1: on.

  The function returns None.

- **`void sm_led( int index, int onoff )`**

  Turns the user LED on or off. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc. **`onoff`**: 0: off, 1: on.

- **`sm_sel00( int index )`**

  Sets the select bits in the write only register on the stepper motor controller expansion board to 00. Updates the shadow register for this latch. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`sm_sel01( int index )`**

  Sets the select bits in the write only register on the stepper motor controller expansion board to 01. Updates the shadow register for this latch. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`sm_sel10( int index )`**

  Sets the select bits in the write only register on the stepper motor controller expansion board to 10. Updates the shadow register for this latch. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`sm_sel11( int index )`**

  Sets the select bits in the write-only register on the stepper motor controller expansion board to 11. Updates the shadow register for this latch. **`index`** is a number from 0 to 15 representing the sequence of stepper motor boards found by **`sm_find_boards`**. The board with the lowest jumper setting will be at position 0, the next lowest at 1, etc.

- **`void sm_int()`**

  Stepper motor controller general interrupt service routine (ISR). Checks the status of all boards listed in the array **`sm_addr`** for an interrupt request (updates **`sm_stat`**). When an interrupt request is detected, the service array flag (**`sm_flags`**) is set, and the stepper motor board is issued a software reset. This reset deactivates the interrupt request emanating from the controller. The service flags are monitored by the master program to determine when an operation has been completed.

  Perform the following steps to exercise this function.

  1. Call **`sm_find_boards`** at the beginning of the program.
  2. Add the following define statement to link this function to the PLCBus ISR.

     ```
     #define USE_MOTORCARD  // activate motor_int isr
     ```

3. Enable the PLCBus interrupt (/AT line) with the following statements at the beginning of the program.

   ```
   relocate_int1();// Relocates the interrupt vector
   outport(ITC,(inport(ITC)|0x02));// Enables IRQ #1
   ```

   Replace the code between the labels **`mirq`** and **`fin`** with your own code to do all motor processing in the background.

- **`void set82adr( int address )`**

  Sends a two-byte address across the PLCBus (for 8 × 2 mode).

- **`void set81adr( int addr )`**

  Sends the last byte of a two-byte address across the PLCBus (for 8 × 2 mode).

IOEXPAND.LIB

These are support functions for the BL1100 expansion boards. They are divided into two classes.

1. Functions that are hard-coded for default base addresses, 0xFxxx.
2. Functions that allow users to specify a board by its node number.

The former class is faster, but is limited to systems with one expansion board; the latter class, therefore, should be limited to multiple expansion board applications.

There is a structure of default addresses to improve lookup speeds for Class 2 functions. There is a structure that holds the default addresses. Instead of specifying a node number (0–3), specify –1. This will load the correct default addresses. The second set of functions allows for stacking of up to four expansion boards on top of the BL1100.

The board addresses are set through jumper J10.

Refer to the ***XP8100 and XP8200 User's Manual*** for the proper board addresses.

- **`int exp_init( int ppia, int ppib, int ppicu, int ppicl )`**

  Initializes the PIO ports of a BL1100 expansion card with the default address of 0xFxxx. The U5 PPI uses mode 0 or the basic I/O mode. **`ppia`**, **`ppib`**, **`ppicu`**,and **`ppicl`** are output values for the PPI output register. Configures Port A as input if **`ppia`** = –1, and Port B as input if **`ppib`** = -1. Configures port C upper nibble as inputs if **`ppicu`** = –1; and port C lower nibble as inputs if **`ppicl`** = –1. All PPI output ports are reset to low when the mode is changed. It is important to output a correct value to the output port right after the mode is changed.

- **`int mux_ch( int chan )`**

  Sets the DG509A multiplexer (U17) of the BL1100 expansion card with the default address of 0xFxxx. **`chan`** is 0 to 3 for (AN0–, AN0+) to (AN3–, AN3+), respectively, to multiplex on (MUX-DA, MUX-DB).

- **`int ad20_mux( int chan )`**

  Sets the multiplexer for the 20-bit AD7703 of the BL1100 expansion card with the default address of 0xFxxx. Channels 0 to 3 select unipolar operation (0 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively, while channels 4 to 7 select bipolar operation (-2.5 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively.

- **`int ad20_rdy()`**

  Tests AD7703 DRDY status from RDTTL bit 1. The function returns 0 if the AD20 is ready, or 1 if AD20 is busy.

- **`int ad20_cal( int mode )`**

  Calibrates the AD7703 on the BL1100 expansion card with the default address of 0xFxxx. Mode 0 calibration does not use the multiplexer. Mode 1 calibration uses the multiplexer to get zero and full scale on Ain. Mux ch0 is the A/D signal to be measured. Mux ch1 is Ain for the Mode 1 first step to calibrate the system offset. Mux ch2 is Ain for Mode 1 second step to calibrate the system gain. Mode 2 calibration uses the current channel to get Ain as zero to calibrate the system offset.

  The following shows the state of SC1 and SC2 during calibration:

Mode	SC1	SC2	Cal type	Zero	FS Steps
0	0	0	self-cal	AGND	REF+1
1	1	1	system offset	Ain	1st of 2
1	0	1	system gain	Ain	2nd of 2
2	1	0	system offset	Ain	REF+1

  The function returns 0, if calibration was completed, or –1, if error during calibration.

- **`long ad20_rd()`**

  Reads 20-bit data from the AD7703 serial data port. The 125-millisecond step response time of AD7703 dictates that a time delay should be guaranteed after a multiplexer switching. A/D data will be valid when DRDY is low for data output at a rate up to 4 kHz. The polarity and channel to read should be set previously with **`ad20_mux`**. Ain ranges from 0 to 2.5 volts for the unipolar mode (PA0 = 0). LSB = 2.5 volts/ 1048576 = 2.384 microvolts. Ain ranges from –2.5 to +2.5 volts for the bipolar mode (PA0 = 1). LSB = 5 volts/1048576 = 4.768 microvolts.

The function returns 20-bit A/D data. For the unipolar mode, 0x00000 = AGND, 0x7FFFF = 1.25 volts and 0xFFFFF = 2.5 volts. For the bipolar mode, 0x00000 = –2.5 volts, 0x7FFFF = AGND and 0xFFFFF = 2.5 volts.

- **`int exp_init_n( int node, int ppia, int ppib, int ppicu, int ppicl, int def )`**

  Initializes the PIO port of a BL1100 expansion card corresponding to the specified node. Node is 0 to 3 for node addresses 0xCxxx to 0xFxxx, respectively. If node equals -1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved. The function returns 0 if initialization is okay, or –1 if an unknown mode is requested.

  Consult the ***XP8100 and XP8200 User's Manual*** for the address configuration.

- **`int get_na( int node, struct node_addr *na )`**

  Gets the node address from the specified node (0–3). The function returns 0 if **`node`** is proper; or –1 if **`node`** is out of range. Node address data are returned in **`struct node_addr *na`**.

- **`int set_def_na( int node )`**

  Sets node address to default node address. The function returns data from **`get_na`**.

- **`int get_def_na( struct node_addr *na )`**

  Gets the default node address. The function returns the node number.

- **`int mux_ch_n( int node, int chan, int def )`**

  Sets DF509A multiplexers on specified BL1100 expansion card node. **`node`** is 0 to 3 for address 0xCxxx to 0xFxxx, respectively. **`chan`** is 0 to 3 for (AN0–, AN0+) to (AN3–, AN3+), respectively. If **`node`** equals –1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as default node in **`def_na`**. If **`def`** equals 0, the node is not saved. The function returns 0 if the **`mux`** setup is okay, or –1 if **`node`** is out of range.

- **`int ad20_mux_n( int node, int chan, int def )`**

  Sets the DG509A multiplexer for the 20-bit AD7703 of a BL1100 expansion card. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. **`chan`** 0–3 selects unipolar operation (0 to 2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively. **`chan`** 4–7 selects bipolar operation (–2.5 to +2.5 volts) for (AN0–, AN0+) to (AN3–, AN3+), respectively. If **`node`** equals –1, the function uses the default address saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved. The function returns 0 if successful, or –1 for invalid **`node`**.

- **`int ad20_rdy_n( int node )`**

  Tests AD7703 DRDY status from RDTTL bit 1 of a specified BL1100 expansion card **`node`**. **`node`** 0–3 specifies the node addresses 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the default node saved in **`def_na`**. The function returns 0 if the AD20 is ready, or –1 if the AD20 is busy or **`node`** is out of range.

- **`int ad20_cal_n( int node, int mode, int def )`**

  Calibrates the AD7703 on a specified BL1100 expansion card. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the node saved in **`def_na`**. If **`def`** equals 1, the node is saved as default node in **`def_na`**. If **`def`** equals 0, the node is not saved. Mode 0 calibration does not use the multiplexer. Mode 1 calibration uses the multiplexer to get zero and full scale on Ain. Mux ch0 is the A/D signal to be measured. Mux ch1 is Ain for the Mode 1 first step to calibrate the system offset. Mux ch2 is Ain for Mode 1 second step to calibrate the system gain. Mode 2 calibration uses the current channel to get Ain as zero to calibrate the system offset. The following shows the state of SC1 and SC2 during calibration.

Mode	SC1	SC2	Cal type	Zero	FS Steps
0	0	0	self-cal	AGND	REF+1
1	1	1	system offset	Ain	1st of 2
1	0	1	system gain	Ain	2nd of 2
2	1	0	system offset	Ain	REF+1

  The function returns 0 if calibration was completed, or –1 if there was an error during calibration.

- **`long ad20_rd_n( int node, int def )`**

  Reads 20-bit data from the AD7703 serial data port. The 125-millisecond step response time of AD7703 dictates that a time delay should be guaranteed after a multiplexer switching. A/D data will be valid when DRDY is low for an output data rate up to 4 kHz. The polarity and channel to read should be set previously with **`ad20_mux`**. Ain ranges from 0 to 2.5 volts for the unipolar mode (PA0 = 0). LSB = 2.5 volts/1048576 = 2.384 microvolts. Ain ranges from –2.5 to +2.5 volts for the bipolar mode (PA0 = 1). LSB = 5 volts/1048576 = 4.768 microvolts. **`node`** 0–3 specifies the node address 0xCxxx to 0xFxxx, respectively. If **`node`** equals –1, the function uses the node saved in **`def_na`**. If **`def`** equals 1, the node is saved as the default node in **`def_na`**. If **`def`** equals 0, the node is not saved.

  The function returns 20–bit A/D data. For the unipolar mode, 0x00000 = AGND, 0x7FFFF = 1.25 volts and 0xFFFFF = 2.5 volts. For the bipolar mode, 0x00000 = –2.5 volts, 0x7FFFF = AGND and 0xFFFFF = 2.5 volts. Returns –1 for an invalid node.

KDM.LIB

These KDM (keyboard/display module) functions provide software drivers for KDM keypads, the text LCD, the graphic LCD, the beeper, and the timer that drives the keypad. The beeper also drives the real-time kernel (RTK) when **`RUNKERNEL`** is defined.

- **`int lk_kxinit()`**

  Initializes variables, buffers and hardware driver associated with servicing the KDM keypad.

- **`int lk_loadtab( int *tab, int tab_size )`**

  Loads keypad numerical table values. Used to rearrange the keypad keys. **`tab`** points to an integer array containing the new keypad arrangement. **`tab_size`** is the table size to change. For example, **`new–table[] = {4,3,2,1,....}`** will rearrange the ordering of the first four keys.

- **`int lk_settab( char flag )`**

  Sets the keypad translate table for keypad sizes greater than 24.

- **`int lk_keyw( char flag )`**

  Writes to specified bits in the key register.

- **`int lk_kxget( char mode )`**

  Gets character from the KDM keypad. If **`mode`** = 0, removes the character from the keypad buffer and returns it. If **`mode !`** = 0, returns the character (if available), but does not remove it from the keypad buffer. The function returns the keypad character pressed, or –1 if the keypad buffer is empty.

- **`int lk_setbeep( int count )`**

  Sets up the variable that is used for the KDM beeper.

- **`int lk_led( int mode )`**

  Turns LEDs on the KDM on/off without conflicting with the keypad driver. **`mode`** = 0 turns off the LEDs. **`mode`** =1 turns on the yellow LED. **`mode`** = 2 turns on the green LED. **`mode`** = 3 turns on both LEDs. The function returns the mode that was passed.

- **`int lk_tdelay( int delay )`**

  Convenient delay mechanism that is tied to **`timer1`** periodic interrupt.

- **`int lk_int_timer1()`**

  Service routine for **`timer1`** interrupt. Drives the beeper and the keypad. Also drives the real-time kernel if **`RUNKERNEL`** is defined.

- **`int lg_init_keypad()`**

  Initializes **`timer1`**, KDM keypad driver and the graphic LCD.

- **`int lk_init_keypad()`**

  Initializes **`timer1`**, keypad driver and the LCD.

- **`void lk_wr( int x )`**

  Writes low byte of **`x`** to LCD register in the high byte of **`x.`**

- **`int lk_rd( int addr )`**

  Reads data from the LCD read register **`addr`**. The function returns the data from LCD read register **`addr`**.

- **`int lk_init()`**

  Initializes LCD on the KDM. Initializes software variables associated with use of the LCD.

- **`int lk_cmd( int cmd )`**

  Sends command in the lower byte of **`cmd`** to the LCD register specified by the upper byte of **`cmd`**.

- **`int lk_wait()`**

  Waits for appropriate LCD unit to clear its busy flag. The function returns 0 or 1 depending on the LCD controller.

- **`int lk_char( char x )`**

  Sends one character to data register of the appropriate LCD.

- **`int lk_ctrl( char x )`**

  Sends one character to control register of the appropriate LCD.

- **`int lk_putc( char x )`**

  Low-level driver (**`printf`** analog) for the LCD. Sends a character to the LCD and updates software variables for storing the LCD screen status.

- **`int lk_nl()`**

  Generates a new line on the LCD screen.

- **`int lk_pos( int line, int col )`**

  Positions LCD cursor to **`line`** and **`col`** location.

- **`int lk_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the LCD. The following escape sequences are available.

  esc p *n mm* positions cursor to line *n* and column *mm.* Example:

  `lk_printf("\x1bp234");`

  means line 2, column 34. Lines are numbered 0, 1, 2, 3. Columns 0,1,..39.

esc 1	Turns cursor on
esc 0	Turn cursor off
esc c	Erases from cursor position to end of line
esc b	Enables blinking cursor mode
esc n	Disables blinking cursor mode
esc e	Erases display and homes cursor

- **`void lk_cgram( char *p )`**

  Special character generator for the LCD. **`*p`** (first byte) is the number of bytes to store (up to 64 for 8 characters). The lower five bits of each byte make one row of the character from left to right and from top to bottom. The eighth row of each is in the cursor position.

- **`int lk_stdcg()`**

  Loads a table of special characters, **`lk_stdchars`**, to the LCD.

- **`int lk_run_menu( char *call_menu, struct lk_menu *menu, int index )`**

  Menuing scheme for the KDM unit. The following mtype codes in the menu structures are available: codes: 0—end of menu, 1—view floating, 2—view floating and adjust in limits, 3—view floating and enter new value on enter, 4—like 2 but call specified function passing pointer after each step, 5—like 3 but call specified function passing pointer to new value, 8—view logical, 9—view logical and adjust true/ false, 10—like 9 but call specified function passing pointer to variable, 16—view date/time, 17—view/ modify date/time, 18—view/ modify date/time and call routine, 20—view time (16-bit), 21—view/modify time (16-bit), 22—view/modify time (16-bit) and then call routine, 32—call a new menu (`msg` is the top line name for new menu, `valptr` is the pointer to the new menu structure, the index is always passed as 0), 40—call a function (`msg` is displayed, `ptr` and `limit` are ignored). The string `call_menu` is initially printed when the menu is entered. The pointer `menu` points to the `lk_menu` structure. The index is the starting point in the menu, often zero. The `run_menu` function returns the last value of the index.

- **`void lk_setdate( char *msg, struct tm *dat )`**

  Sets date data and prints to the LCD. Also prints `msg` to the LCD. Used by `lk–run–menu`.

- **`int lk_chkdat( struct tm *dat )`**

  Checks validity of date data. May change day of the month. The function returns 0 if date data is okay, or 1 for invalid date data.

- **`void lk_showdate( char *msg, struct tm *tmm )`**

  Displays date data and `msg` to the LCD.

- **`uint lk_settime( char *msg, uint time )`**

  Sets time and prints to the LCD. Also prints `msg` to LCD.

- **`int lk_showtime( char *msg, uint time )`**

  Displays `msg` and time data on the LCD.

- **`int st_hour( uint j )`**

  Hour parser used by `lk_run_menu`. The function returns `j`/1800.

- **`int st_min( uint j )`**

  Minutes parser used by `lk_run_menu`. The function returns (`j` mod 1800)/30.

- **`int st_sec( uint j )`**

  Seconds parser used by **`lk_run_menu`**. The function returns
  2 × (`j` mod 30).

- **`uint mk_st( int hour, int min, int sec )`**

  Time data builder used by **`lk_run_menu`**. The function returns
  `hour` × 1800 + **`min`** × 30 + **`sec`** × 2.

- **`uint ad_st( uint t1, uint t2 )`**

  Time data adder used by **`lk_run_menu`**. The function returns adjusted time data of the two times added together.

- **`int lk_secho()`**

  Pulls character from key buffer and generates a short beep.

- **`int lk_lecho()`**

  Pulls character from the keypad buffer and generates a long beep.

- **`void lk_viewl( char *fmt, char var )`**

  Views a logical variable.

- **`float lk_getknum()`**

  Gets a floating-point number from the keypad. The function returns the floating-point number entered through the keypad.

- **`void lg_init()`**

  Initializes the graphic LCD and its associated software variables.

- **`void lg_char( char x )`**

  Writes a character to the graphic LCD.

- **`void lg_putc( char x )`**

  Low-level driver (**`printf`** analog) for the graphic LCD. Puts **`char`** on the graphic LCD and updates software variables that store the graphic LCD screen status.

- **`void lg_nl()`**

  Generates a new line on the graphic LCD screen.

- **`void lg_pos( int line, int col )`**

  Positions cursor on the graphic LCD screen.

- **`void lg_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the graphic LCD. The following escape sequences are available.

  esc p *n mm* positions cursor to line *n* and column *mm.* Example:

  `lg_printf("\x1bp234");`

  means line 2, column 34. Lines are numbered 0, 1, 2, 3. Columns 0,1,..39.

esc 1	Turns cursor on
esc 0	Turn cursor off
esc c	Erases from cursor position to end of line
esc b	Enables blinking cursor mode
esc n	Disables blinking cursor mode
esc e	Erases display and homes cursor

- **`void Set_Display_Mode( int mode )`**

  Sets the display mode of the graphic LCD. **`mode`** is **`DISPLAY_TEXT`** (4) or **`DISPLAY_GRAPHICS`** (8).

- **`void Clear_GrTxt_Screen()`**

  Clears the graphic LCD text screen.

- **`void Stall( int tix )`**

  Software delay loop. Counts down **`tix`** × 10.

- **`void sta01()`**

  Writes 4 to the LCD write register and waits for a 3 on the LCD read register.

- **`void sta03()`**

  Writes 4 to the LCD write register and waits for a 0x08 on the LCD read register.

- **`void lg_wr( int x )`**

  Writes data to graphic LCD register. The register value is in the high byte and data value is in the low byte of **`x`**. Uses **`sta01`** to wait for clear to write.

- **`void lg_wr03( int x )`**

  Writes data to graphic LCD register. The register value is in the high byte and data value is in the low byte of **`x`**. Uses **`sta03`** to wait for clear to write.

- **`void lg_rd()`**

  Waits for clear and reads the graphic LCD read register.

- **`void grp_home_area( char gal, char gah, char ghl, char ghh )`**

  Sets the graphic area by defining the home (**`ghl,ghh`**) and the area (**`gal,gah`**).

- **`void text_home_area( char tal, char tah, char thl, char thh )`**

  Sets the text area by defining the home (**`thl,thh`**) and the area (**`tal,tah`**).

- **`void Graph_Init()`**

  Initializes the graphic LCD text and graphics areas.

- **`void Set_Pointer( int address, int ptr )`**

  Sets the appropriate pointer by using the "pointer set" command. **`address`** is the address to set the pointer to. **`ptr`** is the pointer to set: 1 = cursor, 2 = offset, 4 = address.

  See page 25 of the Toshiba ST-LCD manual.

- **`int Text_Addr( int col, int row )`**

  Computes location of text based on the **`row`** and **`col`** data.

  The function returns

  `GRTXT_BASE_ADDRESS + row × LK_COLS + col` .

- **`void Set_Auto_Mode( int mode )`**

  Sets the graphic LCD into auto mode.

- **`void Set_Overlap_Mode( int mode )`**

  Sets the graphic LCD to overlap mode.

- **`void Define_Cursor( int lines )`**

  Defines the cursor for the graphic LCD.

- **`void Set_Pixel( int col, int row, int wr_mode )`**

  Sets an LCD pixel to coordinates (**`col`**, **`row`**). **`wr_mode`** = 0 to clear, **`wr_mode`** = 1 to set, and **`wr_mode`** = 2 to XOR. (0,0) is the lower left corner. **`col`** ranges from 0 to 239; **`row`** ranges from 0 to 63.

- **`void Clear_Gr_Screen()`**

  Erases the graphic palette by writing 0s to all addresses in the graphic LCD RAM.

- **`void Map_Bit_Pattern( int *config, char *bitarray, int wr_mode )`**

  Maps a bit pattern to the graphic LCD area. **`config`** points to an array of 4-integer data defining the upper left corner (x,y) to start the pattern and the width and height of the figure in dots. **`bitarray`** points to a character data array that has '**`1`**' or '`*`' in each location to set a dot in. Data appear in sequential order, starting at the top left corner, progressing left to right and top to bottom. **`wr_mode`** = 0 to clear; **`wr_mode`** = 1 to set and **`wr_mode`** = 2 to XOR.

- **`void Draw_Line( int stx, int sty, int enx, int eny, int wr_mode )`**

  Draws a line from starting point (**`stx,sty`**) to end point (**`enx,eny`**). **`wr_mode`** = 0 to clear, **`wr_mode`** = 1 to set and **`wr_mode`** = 2 to XOR.

- **`void Draw_Poly( int numpoints, int *point, int wr_mode )`**

  Draws a polygon by connecting successive points. **`numpoints`** is the number of (**`x`**,**`y`**) coordinate pairs. **`point`** points to an integer array of (**`x`**,**`y`**) coordinate pairs.

- **`void Draw_Axis( int ox, int oy, int ex, int ey, int ticks_x, int ticks_y, int wr_mode )`**

  Draws an axis with (**`ox,oy`**) as the axis origin. (**`ex,ey`**) are the highest coordinates of the axis. **`ticks_x`** is the number of x-axis ticks. **`ticks_y`** is the number of y-axis ticks.

- **`void Sin_Wave( int ox, int oy, int ex, int ey, int cycles, int wr_mode )`**

  Draws a sine wave with (**`ox,oy`**) as the sine-wave origin. (**`ex,ey`**) are the highest possible coordinates of the sine wave. **`cycles`** is the number of cycles to display.

LCD2L.LIB

These are LCD support functions. They support the 2 × 20 LCD on all Z-World products that have an LCD port.

- **`void lc_wr( char data )`**

  Low-level routine for writing **`char data`** to a control register of the LCD. The control register accessed is embedded in **`char data`**.

- **`int lc_rd()`**

  Low-level routine to read the LCD register **`LCDWR`**. The function returns the busy flag in bit 7 and the address counter of the LCD in the lower seven bits.

- **`void lc_init()`**

  Initializes the PK2100 or PK2200 LCD by executing the recommended LCD power-up protocol. Sets LCD for auto increment; display and cursor on; and clears the display memory.

- **`int lc_cmd( int cmd )`**

  Waits for LCD busy flag to clear, then sends **`cmd`** to the LCD command register. The function returns 0 if successful in writing to the LCD, or –1 if there is a timeout because the LCD is busy.

- **`int lc_wait()`**

  Waits for the LCD busy flag to clear. The function returns 0 when the LCD busy flag has cleared, or –1 if it times out after ten tries.

- **`void lc_char( char x )`**

  Writes **`char x`** to the LCD data register.

- **`void lc_ctrl( char x )`**

  Writes **`char x`** to the control register of the LCD. Unlike **`lc_wr`**, this function waits for the busy flag of the LCD to clear before writing data to an LCD control register.

- **`int lc_putc( char x )`**

  Decodes **`char x`** for special command sequence for writing to the LCD command or data registers. This function serves as the driver for **`lc_printf`**.

- **`void lc_nl()`**

  Moves the LCD cursor to the first column of the next line. If the current line is the last LCD line, then the cursor position is only moved to column 0 of the current line.

- **`void lc_pos( int line, int col )`**

  Positions PK2100 LCD cursor at the specified **`line`** (0–3) and **`col`** (0–19).

- **`void lc_printf( char *fmt, ... )`**

  This is the **`printf`** analog for the PK2100 LCD.

- **`void lc_cgram( char *p )`**

  Character matrix = 5 rows × 8 cols. **`p`** points to a data array with the following format: first character is the number of bytes to store (8 bytes per character) with a maximum of 64, the lower five bits of each byte form one row of the character from left to right, and the eighth row per special character is in the cursor position.

- **`void lc_stdcg()`**

  Loads eight special characters of arrows and lines to the LCD special character location.

- **`void lcd_init_printf()`**

  Initializes the LCD with **`lcd_init`**. Also initializes related variables to allow for saving duplicate image of the LCD screen.

- **`void lcd_putc( char x )`**

  Decodes **`char x`** for special command sequence for writing to the LCD command or data registers. Serves as the driver for **`lcd_printf`**. Like **`lc_putc`** except that shadow variables for the LCD are also updated.

- **`void lcd_erase()`**

  Erases entire LCD and homes cursor. LCD shadow variables are updated.

- **`void lcd_erase_line( int line )`**

  Erases a specified line on the LCD and updates shadow variables.

- **`void lcd_printf( long cursor, char *fmt, ... )`**

  This is the **`printf`** analog for the LCD screen. Displays a string at a specified starting position and leaves the cursor at a specified end position. **`cursor`** bytes are Y1,X1,Y2,X2, where the most significant byte, Y1, is the start line number (0, 1, 2 or 3); X1 the is start column number (0, 1, 2...), and Y2 and X2 are the final line and column coordinates. The upper four bits of Y2 are used to specify the final state of the cursor (1 = on, 0 = off). Only cursor positioning takes place if **`*fmt`** is a null string.

  When **`lcd_printf`** runs, a semaphore is invoked to ensure that only one execution thread is running through it, so it can be called from various tasks without interference. Execution is suspended for 10 ticks when the semaphore is busy.

  A duplicate copy of the display contents and the cursor location is updated in memory when **`lcd_printf`** prints to the LCD display. The **`lcd_savscrn`** copies this image to a user-specified area. **`lcd_resscrn`** copies the user-saved area back to the screen and the image area. Using these routines, a task can interrupt the current thread and save the current display, use the display in a new thread, and then restore the original display.

- **`void lcd_savscrn( void* s )`**

  Saves LCD screen image to vector identified by **`s`**.

- **`void lcd_resscrn( void* s )`**

  Restores image stored in vector identified by **`s`** to the LCD.

PBUS_LG.LIB

This library contains the PLCBus support functions for the BL1100 controller and the PLCBus interface library for the BL1100 and the BL1300 controllers. The library contains the functions necessary to access PLCBus devices through PIO Port A on the BL1100. The library also provides low-level PLCBus functions as well as high-level functions for the relay and DAC expansion boards.

The bus must interface to the PIO port as follows:

PIO pin 0: STB	PIO pin 4: D2
PIO pin 1: A3	PIO pin 5: D3
PIO pin 2: A2	PIO pin 6: D0
PIO pin 3: A1	PIO pin 7: D1

- **`void PBus12_Addr( int addr )`**

  Sets the current address for the PLCBus . All read and write operations will access this address until a new address is set. **`addr`** is the 12-bit physical address with the first and third nibbles swapped (most significant nibble in the lower four bits).

- **`void PBus4_Write( char data )`**

  Writes 4-bit data on PLCBus . The address must be set by a call to **`PBus12_Addr`** before calling this function. **`data`** should contain the value to write in the lower four bits.

- **`int PBus4_Read0()`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD0`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_Read1()`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD1`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_ReadSp()`**

  Reads 4 bits of data from the PLCBus using a **`BUSSPARE`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns PLCBus data in the lower four bits (the upper bits are undefined).

- **`int Relay_Board_Addr( int board )`**

  Converts a logical relay board address to a physical PLCBus address. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form `pppzyx` where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**000**`y`**000**`z`**.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Set_PBus_Relay( int board,int relay,`**
  `int state )`

  Sets a relay on an expansion bus relay board. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). **`relay`** is the relay number on the board (0–5 for XP8300 board; 0–7 for XP8400 board). **`state`** must be 1 to turn the relay on and 0 to turn the relay off.

- **`int DAC_Board_Addr( int bd )`**

  Converts a logical DAC board address to a physical PLCBus address. **`bd`** must be a number between 0 and 63, and represents the DAC board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J3 on the board. `ppp` values of 000, 001, 010, etc., correspond to PAL numbers of FPO4800, FPO4810, FPO4820, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J3 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**001`y`000`z`.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Write_DAC1( int val )`**

  Loads Register A of DAC #1 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC1`** is called.

- **`void Write_DAC2( int val )`**

  Loads Register A of DAC #2 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC2`** is called.

- **`void Latch_DAC1()`**

  Moves the value from Register A of DAC 1 to the Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC1`**.

- **`void Latch_DAC2()`**

  Moves the value from Register A of DAC #2 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC2`**.

- **`void Init_DAC()`**

  Initializes DAC board and sets all output values to 0. Call this function before writing data to the DAC. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC1( int val )`**

  Sets DAC #1 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC2( int val )`**

  Sets DAC #2 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void DAC_On()`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void DAC_Off()`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void Reset_PBus()`**

  Resets the PLCBus.

- **`int Poll_PBus_Node( int addr )`**

  Polls a PLCBus device by performing a **`BUSRD0`** cycle and checking the low bit of the returned value. **`addr`** is the 12-bit physical address of the device, with the first and third nibbles swapped.

  The function returns 1 if **`node`** answers poll, 0 if not.

- **`void Reset_PBus_Wait()`**

  Provides the minimum delay necessary for PLCBus expansion boards after a bus reset, assuming a 9-MHz CPU. This delay will be insufficient for a faster CPU and must be increased.

PBUS_TG.LIB

These functions support the BL1000 controller. The PLCBus interface library is provided for the BL1000. This library contains functions necessary to access PLCBus devices through PIO Port B on the BL1000. The library provides low-level PLCBus functions as well as high-level functions for the relay and DAC expansion boards.

The bus must interface to the PIO port as follows.

PIO pin 0: D1	IO pin 4: A1
PIO pin 1: D0	PIO pin 5: A2
PIO pin 2: D3	PIO pin 6: A3
PIO pin 3: D2	PIO pin 7: STB

- **`void PBus12_Addr( int addr )`**

  Sets the current address for the PLCBus. All read and write operations will access this address until a new address is set. **`addr`** is the 12-bit physical address with the first and third nibbles swapped (most significant nibble in the low four bits).

  The function returns None.

- **`void PBus4_Write( char data )`**

  Writes 4-bit data on the PLCBus. The address must be set by a call to **`PBus12_Addr`** before calling this function. **`data`** should contain the value to write in the lower four bits.

- **`int PBus4_Read0()`**

  Reads 4 bits of data from the PLCBus using a `BUSRD0` cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_Read1()`**

  Reads 4 bits of data from the PLCBus using a **`BUSRD1`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int PBus4_ReadSp()`**

  Reads 4 bits of data from the PLCBus using a **`BUSSPARE`** cycle. The address must be set by a call to **`PBus12_Addr`** before calling this function. The function returns the PLCBus data in the lower four bits (the upper bits are undefined).

- **`int Relay_Board_Addr( int board )`**

  Converts a logical relay board address to a physical PLCBus address. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**000**`y`**000**`z`**.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Set_PBus_Relay( int board,int relay, int state )`**

  Sets a relay on an expansion bus relay board. **`board`** must be a number between 0 and 63, and represents the relay board to access. This number has the binary form **`pppzyx`**, where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J1 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4500, FPO4510, FPO4520, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J1 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). **`relay`** is the relay number on the board (0–5 for XP8300 board; 0–7 for XP8400 board). **`state`** must be 1 to turn the relay on and 0 to turn the relay off.

- **`int DAC_Board_Addr( int bd )`**

  Converts a logical DAC board address to a physical PLCBus address. **`bd`** must be a number between 0 and 63, and represents the DAC board to access. This number has the binary form **`pppzyx`** where **`ppp`** is determined by the board PAL number and **`x`**, **`y`**, and **`z`** are determined by jumper J3 on the board. **`ppp`** values of 000, 001, 010, etc., correspond to PAL numbers of FPO4800, FPO4810, FPO4820, etc.; **`x`**, **`y`**, and **`z`** correspond to jumper J3 pins 1-2, 3-4, and 5-6, respectively (0 = closed, 1 = open). The resulting address is in the form **`pppx`**001**`y`**000**`z`**.

  The function returns the PLCBus address of the board specified, with the first and third nibbles swapped; this address may be passed directly to **`PBus12_Addr`**.

- **`void Write_DAC1( int val )`**

  Loads Register A of DAC #1 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC1`** is called.

- **`void Write_DAC2 ( int val )`**

  Loads Register A of DAC #2 with the given 12-bit value. The board address must have been set previously with a call to **`PBus12_Addr`**. The value in **`val`** will not actually be output until **`Latch_DAC2`** is called.

- **`void Latch_DAC1 ()`**

  Moves the value in Register A of DAC #1 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC1`**.

- **`void Latch_DAC2 ()`**

  Moves the value in Register A of DAC #2 to Register B. The value in Register B represents the actual DAC output. The board address must have been set previously with a call to **`PBus12_Addr`**, and the value should have been loaded into Register A with a call to **`Write_DAC2`**.

- **`void Init_DAC()`**

  Initializes DAC board and sets all output values to 0. Call this function before writing data to the DAC. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC1( int val )`**

  Sets DAC #1 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void Set_DAC2( int val )`**

  Sets DAC #2 to the value specified in the lower 12 bits of **`val`**. In voltage-output mode (J1 pins 2-3 jumpered), V_{OUT} = (**`val`**/4096) × 10.22 volts with Z-World default settings. In current-output mode (J1 pins 1-2 jumpered), I_{OUT} = (**`val`**/4096) × 22 milliamps with Z-World default settings. The board address must have been set previously with a call to **`PBus12_Addr`**.

- **`void DAC_On()`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void DAC_Off()`**

  Controls the high-side switch activation line. Only used with switch option U10-LT1188.

- **`void Reset_PBus()`**

  Resets the PLCBus.

- **`int Poll_PBus_Node( int addr )`**

  Polls a PLCBus device by performing a **`BUSRD0`** cycle and checking the low bit of the returned value. **`addr`** is the 12-bit physical address of the device, with the first and third nibbles swapped. The function returns 1 if **`node`** answers poll, 0 if not.

- **`void Reset_PBus_Wait()`**

  Provides the minimum delay necessary for PLCBus expansion boards after a bus reset, assuming a 9-MHz CPU. This delay will be insufficient for a faster CPU and must be increased.

Appendix A: Dynamic C Libraries

The libraries described in Chapter 1 include standard C string and math functions in addition to general support functions specific to Z-World’s controllers.

Dynamic C's function libraries provide a way to bring in only those portions of system code that a particular program uses. The file **`LIB.DIR`** contains a list of all libraries known to Dynamic C. This list may be modified by the user. In particular, any library created by a user must be added to this list.

Libraries are "linked" with a user's application through the **#use** directive. Files identified by **`#use`** directives are nestable, as shown in Figure A-1.

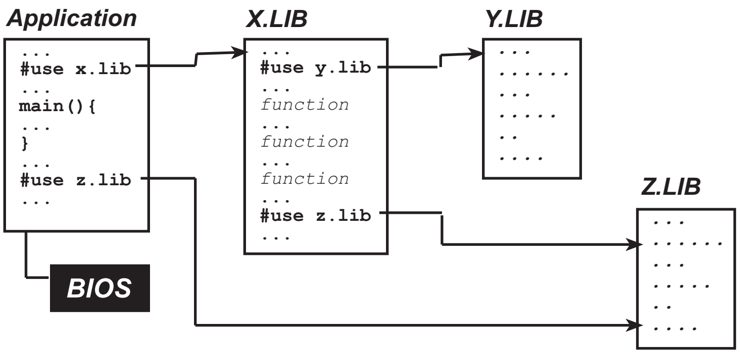

Figure A-1. Linking Nestable Files in Dynamic C

The file **`DEFAULT.H`** contains several lists of libraries to **`#use`**, one list for each product that Z-World ships. Dynamic C usually knows which controller is being used, so it selects the libraries appropriate to that controller. These lists are the *defaults*. A programmer may find it convenient or necessary to add or remove libraries from one or more of the lists.

The default libraries for a Z-World controller contain many function names, global variable names, and in particular, many macro names. It is likely that a programmer may try to use one of the Z-World names for a newly written program. Unpredictable problems can arise. Z-World recommends that **`DEFAULT.H`** be edited to comment out libraries that are not needed.

Table A-1 lists the libraries included with Dynamic C. Other libraries, **`LSTAR.LIB`**, **`MICROG.LIB`**, **`LGIANT.LIB`**, **`RG.LIB`**, **`SCOREZ1.LIB`**, **`LPLC.LIB`**, and **`PS.LIB`** exist only for backward compatibility.

Table A-1. Libraries Included with Dynamic C

`5KEY.LIB`	The basic “five-key system” for the PK2100 series and PK2200 series controllers.
`5KEYEXTD.LIB`	Extensions to the “five-key system.”
`96IO.LIB`	Driver functions for the BL100’s DGL96 daughter board.
`AASC.LIB`	Abstract Asynchronous Serial Communication functions.
`AASCDIO.LIB`	STDIO-specific routines supporting the AASC library.
`AASCSCC.LIB`	SCC-specific routines to support the AASC library. The SCC is the Zilog 85C30 Serial Communication Controller.
`AASCUART.LIB`	XP8700 series support for the AASC library. The XP8700 is formerly the RS232 PLCBus expansion board.
`AASCZ0.LIB`	Z0-specific routines to support the AASC library. Z0 is the Z180 ASCI Serial Port 0.
`AASCZ1.LIB`	Z1-specific routines to support the AASC library. Z1 is the Z180 ASCI Serial Port 1.
`AASCZN.LIB`	ZNet-specific routines to support the AASC library.
`BIOS.LIB`	Contains prototypes of functions and declarations of variables defined in, and used by, the BIOS.
`BL1000.LIB`	Functions for the BL1000.
`BL11XX.LIB`	Functions for the BL1100.
`BL12XX.LIB`	*Empty library.*
`BL13XX.LIB`	Functions for the BL1300.
`BL14_15.LIB`	Functions for the BL1400 series and BL1500 series controllers.
`BL16XX.LIB`	Functions for the BL1600.
`CIRCBUF.LIB`	Abstract data type functions for circular buffers (used by the AASC driver).
`CM71_72.LIB`	Functions for the CM7100 series and CM7200 series core modules. These are formerly the SmartCore Z1 and Z2.
`CPLC.LIB`	Functions for PK2100, PK2200, and BL1600.

continued…

Table A-1. Libraries Included with Dynamic C (continued)

`DC.HH`	This file contains definitions basic to, and required by, Dynamic C. This file is required.
`DEFAULT.H`	Contains lists of **`#use`** directives for various Z-World controllers. Dynamic C automatically selects the list appropriate for controller being programmed.
`DMA.LIB`	Support functions for the Z180 on-chip DMA (direct memory access) channels.
`DRIVERS.LIB`	Driver functions for some hardware devices.
`EZIO.LIB`	Driver functions for a board-independent unified I/O space.
`EZIOCMMN.LIB`	Common definitions for all **`EZIO…`** libraries.
`EZIOPBDV.LIB`	PLCBus device drivers supporting the EZIO library.
`EZIOPK23.LIB`	PK2300 function support for the EZIO library.
`EZIOPLC.LIB`	PLCBus functions for boards that have native PLCBus ports (BL1200 series, BL1600 series, PK2100 series, and PK2200 series.
`FK.LIB`	New "five-key system" support for the PK2100 series and PK2200 series controllers. They are to be used with cooperative multitasking (i.e., costatements).
`IOEXPAND.LIB`	Driver functions for BL1100 series daughter boards.
`KDM.LIB`	Driver functions for Z-World KDMs (keyboard/display modules).
`LCD2L.LIB`	Two-line LCD support for the PK2100 series and PK2200 series controllers.
`MATH.LIB`	Useful mathematical and trigonometric functions.
`MISC.LIB`	Miscellaneous functions for KDM support.
`MODEM232.LIB`	Modem functions for the PK2100 series and PK2200 series controllers. Used with **`Z0232.LIB`**, **`S0232.LIB`**, **`XP87XX.LIB`**, **`NETWORK.LIB`** and **`SCC232.LIB`**.
`NETWORK.LIB`	Opto22 9-bit binary protocol to support master-slave networking. Uses ASCI port 1 of the Z180.
`PBUS_LG.LIB`	Functions that operate the PLCBus with a BL1100.
`PBUS_TG.LIB`	Functions that operate the PLCBus with a BL1000.

continued…

Table A-1. Libraries Included with Dynamic C (concluded)

PK21XX.LIB	Functions for the PK2100.
PK22XX.LIB	Functions for the PK2200.
PLC_EXP.LIB	PLCBus functions for boards that have native PLCBus ports (BL1200 series, BL1600 series, PK2100 series, and PK2200 series).
PRPORT.LIB	Functions that implement a parallel port communication protocol between a controller and a PC.
PWM.LIB	Pulse-width modulation functions.
RTK.LIB	Real-time kernel (RTK).
S0232.LIB	Serial communication driver for SIO port 0 on the BL1100 series controller.
S1232.LIB	Serial communication driver for SIO port 1 on the BL1100 series controller.
SCC232.LIB	Serial communication driver for the ports on the SCC chip, Zilog's 85C30 Serial Communication Controller.
SRTK.LIB	Simplified real-time kernel for all controllers.
STDIO.LIB	Functions relating to the **STDIO** window in Dynamic C.
STRING.LIB	This file contains functions for manipulating strings.
SYS.LIB	General system functions.
VDRIVER.LIB	Virtual driver functions (for all controllers).
XMEM.LIB	Functions for moving information to and from extended memory, as well as other functions (such as address computation) related to extended memory.
XP82XX.LIB	Driver functions for the XP8200 series PLCBus board.
XP87XX.LIB	Serial communication functions for an XP8700 series PLCBus board.
XP87XX2.LIB	Serial communication functions that support a second XP8700 (see **XP87XX.LIB** below).
XP88XX.LIB	Functions for the XP8800 series PLCBus device.
Z0232.LIB	Serial communication driver for Z0. Z0 is the Z180 ASCI Serial Port 0.
Z1232.LIB	Serial communication driver for Z1. Z1 is the Z180 ASCI Serial Port 1.
ZNPAKFMT.LIB	Lower level functions supporting the ZNet.

APPENDIX B: USING AASC LIBRARIES

The Abstract Application-Level Serial Communication (AASC) library and its low-level support functions facilitate serial communication between controllers and between a controller and another device such as a PC.

AASC Library Description

AASC libraries allow the programmer to create buffered character streams that perform input/output to/from ports in the communication devices. One principal library, **`AASC.LIB`**, contains all the functions required for these tasks. Table B-1 lists the support libraries used with **`AASC.LIB`**.

Table B-1. Drivers Used in AASC.LIB

Driver Library	Description
`AASCDIO.LIB`	Contains specific standard input/output (**`STDIO`**) routines to support the AASC libraries.
`AASCSCC.LIB`	Operates channels on the Zilog 85C30 Serial Communication Controller used in BL1100 and BL1700 controllers.
`AASCUART.LIB`	Operates RS-232 port on the XP8700 PLCBus expansion board supported by most Z-World controllers.
`AASCURT2.LIB`	Operates RS-232 port on the XP8700 PLCBus expansion board on controllers (e.g., BL1700) with 16-bit PLCBus addressing.
`AASCZ0.LIB`	Handles communication on the Z0 port of the Zilog Z180 microprocessor used by Z-World controllers. This port is usually connected to an RS-232 driver.
`AASCZ1.LIB`	Handles communication on the Z1 port of the Zilog Z180 microprocessor used by Z-World controllers. This port is usually connected to an RS-485 driver.
`AASCZN.LIB`	Operates ZNet-specific routines on the RS-485 network. All participating controllers must use the same driver. One controller is designated the *master controller* by defining the macro **`ZNMASTER`** to be non-zero before invoking **`#use AASCZN.LIB`**. This library uses the Z1 port of the Zilog Z180 microprocessor.

The AASC libraries are as device-independent as possible. Programs include only the **`AASC.LIB`** code and the code required for the communication devices used by the application (for example, **`AASCSCC.LIB`**). The application handles different communication devices simply by creating separate device channels.

Two hidden circular buffers for each AASC channel store incoming and outgoing information. This allows the application to process incoming and outgoing information in chunks not larger than the circular buffers. The buffer size is specified in the application.

AASC support libraries implement custom device drivers and interrupt service routines (ISRs) for each communication device. The application only needs to initialize a channel and a local buffer, then make function calls to check the status of the buffers, and read or write to/from the buffers.

AASC Library Operation

AASC libraries read (receive), write (transmit), peek (search), provide status, and handle errors. Figure B-1 shows the hierarchy of these AASC functions. Note that the management of the circular buffer and the hardware/serial ISR levels is hidden from the programmer. These two reserved levels are contained in the support libraries listed in Table B-1.

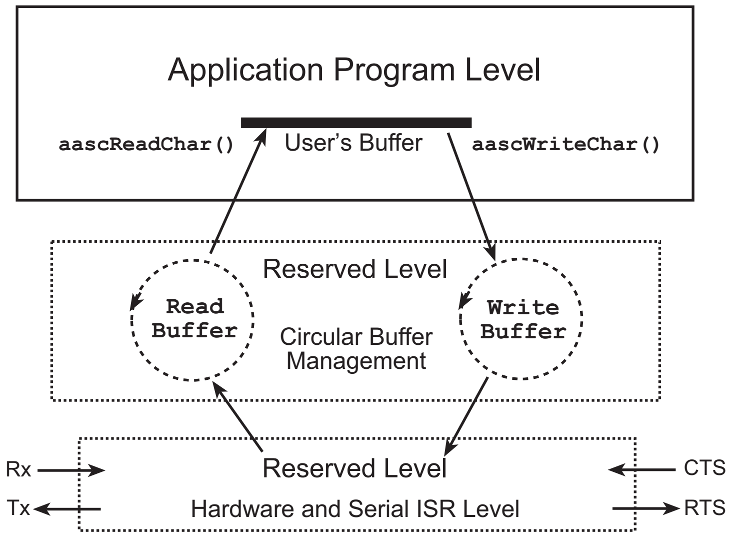

Figure B-1. Hierarchy of AASC Functions

Read

Information is received either by block or by byte. Only one method is needed, but the other can always be implemented. It is more efficient to have both methods available. The block read function supports fixed-size and variable-size reads. The application may read exactly *n* bytes, it may read nothing at all, or it may read up to *n* bytes. In any case, the function returns the number of bytes actually read.

Read operations may preempt write operations and vice versa, but a read operation cannot preempt another read operation and a write operation cannot preempt another write operation.

Write

The transmit (write) routines are mirror images of the read functions. There is one function for byte writes and one for block writes. The block write function can write part of a block, or it may write all or none of the block. This is important for multi-threaded programs because writing all or none prevents interleaving messages originating from different cooperative threads.

Peek

A special function supported by the AASC libraries allows the application to "peek" into the buffer without retrieving a byte. The peek function **`aascPeek`** searches for a substring, for example, to identify the type of incoming packet, without actually changing the contents of the buffer. Another "peek" type function, **`aascScanTerm`**, can also search for a particular character such as the terminating character of a packet.

Status and Errors

AASC libraries provide full status reports about the application. The libraries can report the number of bytes used and the number of bytes still free in the read or write buffers. Such information is useful for the application to schedule message checking or dynamic transmission.

AASC libraries also report both hardware errors (for example, framing error, parity error) and software errors (for example, buffer overrun). Error conditions are not cleared automatically.

Library Use

Follow these six steps when using AASC libraries.

1. Identify the communication device (e.g., Z0, SCC Channel A, UART).
2. Allocate and initialize the channel with **`aascOpen()`**.
3. Set up read (receive) and write (transmit) circular buffers (e.g., use **`aascSetReadBuf()`**).
4. Carry out reads and writes (e.g., use **`aascWriteChar()`**).
5. Check status and handle errors (e.g., use **`aascGetError()`**).
6. When finished, close the channel with **`aascClose()`**.

Sample Program

The following sample program provides an example of the use of the AASC framework in asynchronous serial communication with a terminal. The program demonstrates how to use port SCC Channel A as an AASC device. Other sample programs may be found in the Dynamic C **`SAMPLES\AASC`** subdirectory.

This program simply echoes text typed at an ascii terminal back to the terminal. Connect a controller with a serial communication controller IC through SCC Channel A to a PC or dumb terminal. If using a PC, Windows **terminal.exe** can be used in **ANSI Terminal Emulation** with **Local Echo** disabled and **Flow Control** set to **None**. If RTS/CTS handshaking is enabled by setting the macro **SHAKE** to non-zero, enable **Flow Control**" within **terminal.exe** to **Hardware**. This sample program defaults to settings of **No Parity**, **One Stop Bit**, and **Eight Data Bits**. Set your PC accordingly.

The following steps describe how this "echoing" process works.

1. The program accesses **AASC.LIB** and the appropriate AASC library **AASCSCC.LIB** with **#use**.
2. Definitions are created for circular read and write buffers, and for the user buffer **workBuffer**. A user buffer pointer, **pworkBuffer**, is also created for this example.
3. **_GLOBAL_INIT()** is called to initialize the AASC framework.
4. The function **aascOpen()** is used to create a channel to the **DEV_SCC** device at 8N1.
5. The program checks to make sure that a controller with an SCC IC is being used.
6. The transmitter and receiver for the channel **chan** are switched on by **aascTxSwitch()** and by **aascRxSwitch()**.
7. The program sets up the circular buffers with **aascReadBuf** and **aascSetWrite Buf**.
8. If a character is read, the program enters another loop that sends the characters in **workBuffer** back to the remote terminal. The function will not return until all the characters are read from **workBuffer** and sent back to the terminal. (For example, if two characters are in **workBuffer**, the function will return only when both characters are sent.)

SCCECHO.C

```
#use aasc.lib
#use aascscc.lib
#define BUFSIZE 684    // Size of circular buffer.
#define BAUDMULT 8     // multiples of 1200 bps
                       // (8 × 1200 bps = 9600 bps).
#define SHAKE 0   // Set to 1 for RTS/CTS handshaking.
char readBuffer[BUFSIZE],writeBuffer[BUFSIZE];
char workBuffer[BUFSIZE],*pworkBuffer;
struct _Channel *aascChannel;
main(){
  _GLOBAL_INIT();   // This must be the first action
                    // performed in main().
  // Open channel A of the SCC at 8N1
  aascChannel = aascOpen( DEV_SCC, SHAKE,
    SCC_A | SCC_1STOP | SCC_NOPARITY | SCC_8DATA |
    SCC_1200*BAUDMULT, NULL);
  if(aascChannel==NULL) {
    printf("SCC channel A not available.");
    return;
  }
  // Set up the circular buffers.
  aascSetReadBuf( aascChannel, readBuffer,
    sizeof( readBuffer));
  aascSetWriteBuf( aascChannel, writeBuffer,
    sizeof( writeBuffer));
  // Process the data transfer.
  while(1) {
    hitwd();
  // Perform data transfer.
    if( aascReadChar( aascChannel, workBuffer) ) {
      while( !aascWriteChar( aascChannel,
        workBuffer[0]) ) {
        hitwd();
      }
    }
  }
}
```

XModem Transfer

The AASC libraries have extensive support for the **`XModem-CRC`** transfer protocol. The AASC libraries allow the application to define callback functions to read or write each block of an XModem packet. This means there is no need to have the entire transfer block ready before transmission, or to allocate space for the entire incoming block. Default callback functions are provided for normal read-to-memory or write-from-memory operations.

Library Use

1. Initialize the virtual driver.
2. Initialize the AASC framework with an appropriate device such as SCC Channel A.
3. Initialize an XModem data buffer and the number of bytes to transfer with **`aascXMWrInitPhy()`** or **`aascXMRdInitPhy()`** for physical memory, or **`aascXMWrInitLog()`** or **`aascXMRdInitLog()`** for logical memory.
4. Initialize XModem transfer with **`aascWriteXModem()`** or **`aascReadXModem()`**.
5. Perform the XModem transfer with **`aascWriteModem()`** or **`aascReadXModem()`**.

Sample Program

The following sample program provides an example of the use of an AASC framework in XModem data transfer. The program sends one block of 128 characters to a remote device using **`XModem-CRC`**. Configure the remote device for 9600 bps at 8N1 without RTS/CTS flow control.

The virtual driver must be used since XModem incorporates costatements to enable multitasking.

Note that any channel may be used by changing SCC Channel A to the desired port. For example, to use port Z1 on the Z180, change **`AASCSCC.LIB`** to **`AASCZ1.LIB`**, and change the parameters in **`aascOpen()`** to reflect those for Z1.

The following steps describe the XModem transmission example.

1. The program accesses the appropriate libraries with **`#use`**.
2. Definitions are created for the circular read and write buffers, and for the XModem buffer.
4. **`aascInit()`** is called to initialize the AASC framework.
5. A data string is created for transfer.
6. **`VdInit()`** is called to initialize the virtual driver.
7. **`aascOpen()`** is used to create a channel to the **`SCC_A`** device at 8N1 and 9600 bps.
8. The program checks for the presence of the SCC chip on the controller.
9. The circular buffers are then initialized by **`aascSetReadBuf()`** and by **`aascSetWriteBuf()`**, and are made accessible to the AASC framework.
10. XModem transmission is then performed by repeatedly calling **`aascWriteXModem()`** with the initialization parameter set to 0.
11. XModem transmission finishes when **`aascWriteXModem()`** returns a 1.

XM_SEND.C

```
#use vdriver.lib
#use aasc.lib
#use aascscc.lib
#define BUFSIZE 1024  // Size of circular buffer.
#define BAUDMULT 8    // multiples of 1200 bps
                      // (8 × 1200 bps = 9600 bps).
struct _Channel *aascChannel;
char circBufIn[BUFSIZE], circBufOut[BUFSIZE];
char aascBuffer[BUFSIZE];
int aascInit(void);
void main(void){
  // Initialize the AASC framework.
  if( !aascInit() ) exit(-1);
  // Create some data to transfer.
  strcpy( aascBuffer, "This is some xmodem data transfer…");
  // Process the data transfer.
  while(1) {
    hitwd();
    printf("Press any key to initiate Xmodem
      Controller-to-Device transfer.\r");
    hitwd();
    if( kbhit() ) {
      getchar();
      printf("\n\nXmodem transfer initiated...\n");
      hitwd();
      // Set up XModem transfer to logical memory.
      aascXMWrInitLog( (unsigned) aascBuffer, 128);
      aascWriteXModem( aascChannel, 0, 1,
        aascWrCallBackLg );
      while( !aascWriteXModem( aascChannel, 0, 0,
          aascWrCallBackLg ) ) hitwd();
      printf("\n\nXmodem transfer finished...\n\n");
      hitwd();
    }
  }
}
```

continued...

```
int aascInit(void){
  // Initialize the virtual driver
  VdInit();
  // Open channel A of the SCC at 8N1
  aascChannel = aascOpen( DEV_SCC, 0,
    SCC_A | SCC_1STOP | SCC_NOPARITY | SCC_8DATA |
      SCC_1200*BAUDMULT, NULL);
  if(aascChannel==NULL) {
    printf("SCC channel A not available.");
    return;
  }
  // Set up the circular buffers.
  aascSetReadBuf( aascChannel, circBufIn,
    sizeof(circBufIn) );
  aascSetWriteBuf( aascChannel, circBufOut,
    sizeof(circBufOut) );
}
```

APPENDIX C: ***Z-WORLD PRODUCTS***

Name	Description
PK2300	9.216-MHz packaged controller. Provides 19 digital I/O lines (11 lines are configurable), 2 serial channels, a resistance measurement input, and real-time clock. ABS enclosure.
PK2310	PK2300, without RTC and resistance measurement circuit.
PK2200	18.432-MHz packaged controller. Provides 16 digital inputs, 14 high-current outputs, 2 serial channels, and enclosure with 2x20 LCD and 2x6 keypad.
PK2210	PK2200, with 9.216-MHz clock.
PK2220	PK2200 without enclosure, LCD or keypad.
PK2230	PK2200 with a 9.216-MHz clock. No enclosure, LCD or keypad.
PK2240	PK2200 with a 128 x 64 EL backlit graphic LCD and 3 x 4 keypad.
PK2100	6.144-MHz packaged controller. Provides 7 digital inputs, 10 high-current outputs, 6 universal inputs, 2 SPST relays, 2 serial channels, one high-gain analog input, 2 analog outputs, and a rugged enclosure with 2x20 LCD and 2x6 keypad. Operates at 24 volts. D.C.
PK2110	PK2100 that operates at 12 V D.C.
PK2120	PK2100 without enclosure, LCD or keypad.
PK2130	PK2120 that operates at 12 V D.C.
BL1600	9.216-MHz board-level controller. Provides 12 digital inputs, 14 digital outputs, 2 serial channels, EEPROM and real-time clock
BL1610	BL1600 without serial channels, high-current drivers, EEPROM, or real-time clock.
BL1500	9.216-MHz board-level controller. Provides 24 PIO lines, four 12-bit ADC channels, one RS232 channel, one RS485 channel, and real-time clock. 128K SRAM.
BL1510	BL1500 with 32K SRAM. No real-time clock. Provides 2 additional PIO lines.
BL1520	BL1500 with 32K SRAM. No real-time clock or 12-bit A/D converter. Provides 2 additional PIO lines.
BL1400	6.144-MHz board-level controller. Provides 12 PIO lines, one RS-232 channel, one RS-485 channel and real-time clock.
BL1410	BL1400 without the RS-485 channel and real-time clock. Provides 2 additional PIO lines.

Name	Description
BL1300	9.216-MHz board-level controller. Provides 4 serial channels and two 16-bit parallel ports. Optional enclosure.
BL1200	9.216-MHz board-level controller. Provides 8 optically isolated inputs, 6 high-current outputs, and 2 RS-485 channels.
BL1100	9.216-MHz board-level controller. Provides 16 digital I/O lines, 8 high-current drivers, 7 10-bit ADC inputs, 2 RS-232 channels and 2 RS-485 channels. Switching power supply.
BL1110	BL1100 with a linear (not switching) power supply.
BL1120	BL1100 with a 12.288-MHz clock and linear (not switching) power supply. Runs 50% faster.
CM7100	18.432-MHz microprocessor core module. Provides processor, 384 device addresses, 128K SRAM, EEPROM, real-time clock, and 691 supervisor.
CM7110	CM7100 with 9.216-MHz clock.
CM7120	CM7100 with 9.216-MHz clock and 32K SRAM.
CM7130	CM7100 with 9.216-MHz clock and 32K SRAM. Without 691 supervisor, real-time clock, and EEPROM.
CM7200	18.432-MHz microprocessor core module. Provides processor, 384 device addresses, 128K SRAM, real-time clock, 691 supervisor, and 128K flash EPROM.
CM7210	CM7200 with 9.216-MHz clock.
CM7220	CM7200 with 9.216-MHz clock and 32K SRAM.
CM7230	CM7200 with 9.216-MHz clock and 32K SRAM. Without 691 supervisor or real-time clock.

Other products include the BL1000, LP3100 and the PK2400.

INDEX

Z-World

2900 Spafford Street
Davis, California 95616-6800 USA

Telephone:	(530) 757-3737
Facsimile:	(530) 753-5141
24-Hour FaxBack:	(530) 753-0618
Web Site:	http://www.zworld.com
E-Mail:	zworld@zworld.com

Part No. 019-0002-03
Revision 3

Printed in U.S.A.